Ein Framework zur Berechnung der Hermite-Normalform von großen, dünnbesetzten, ganzzahligen Matrizen

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)

von
Diplom Informatiker
Patrick Theobald
aus Wadern

Referenten:	Prof. Dr. J. Buchmann
	Prof. Dr. W. Bosma
Tag der Einreichung:	09. November 2000
Tag der mündlichen Prüfung:	12. Dezember 2000

Darmstadt, 2001
D 17

Zusammenfassung

Unter der Hermite–Normalform (abgekürzt HNF) einer ganzzahligen Matrix A versteht man (vereinfacht dargestellt) eine zu A äquivalente obere Dreiecksmatrix H, bei der die Einträge oberhalb der Diagonalen modulo dem zugehörigen Diagonalelement reduziert sind, d.h. betragsmäßig zwischen Null und dem jeweiligen Diagonalelement liegen. Eine Matrix A ist zur einer Matrix H genau dann äquivalent, wenn eine ganzzahlige, unimodulare Matrix T existiert, so daß gilt $A \cdot T = H$.

Während der letzten Jahre wurden eine Vielzahl von Algorithmen zur Berechnung der HNF entwickelt, angefangen bei den Algorithmen von Charles Hermite[Her51] bis hin zu den Algorithmen von George Havas [HM94a].

In der Praxis wird man bei der Berechnung der HNF von großen, ganzzahligen Matrizen auf einer speziellen, festgelegten Maschine mit dem Problem konfrontiert, daß während der Durchführung der Berechnung zunächst die Eintragsdichte, d.h. der prozentuale Anteil der von Null verschiedenen Elemente unter den Einträgen, und dann die Eintragsgröße der entstehenden Zwischeneinträge in unkontrollierter Art und Weise anwachsen. Dies führt zu einem hohen Speicherplatzverbrauch und mündet schließlich im Abbruch der Berechnung.

Um diesem Phänomen zu begegnen, wurden vorrangig für dichtbesetzte Matrizen eine Vielzahl von optimierten Algorithmen entwickelt [Coh93, Dom89, DKJ87, Hav91]. Für die Matrizen, die im Kontext der Klassengruppenberechnung entstehen, (vgl. [Jr.99, Nei01]) reichen diese Optimierungen nicht aus, d.h. es ist nicht möglich, auf der uns zur Verfügung stehenden Hardware die HNF dieser Matrizen zu berechnen. Vor dem Hintergrund dieser Problemstellung ist die vorliegende Arbeit entstanden.

Die Idee dieser Arbeit basiert auf der Beobachtung, daß sich unterschiedliche “einfache” (aus der Literatur bekannte) HNF–Algorithmen angewendet auf die gleiche Eingabematrix in Bezug auf ihr Laufzeitverhalten, d.h. die Entwicklung des Speicherplatzverbrauchs, die Entwicklung der Eintragsdichte, die Entwicklung der Eintragsgröße u.a., unterschiedlich verhalten. Wenn also nicht ein einzelner der “einfachen” HNF–Algorithmen in der Lage ist, die HNF einer dünnbesetzen, ganzzahligen Matrix unter Berücksichtigung der zur Verfügung stehenden Hardwareressourcen zu berechnen, dann könnte es einer geeigneten Kombination dieser Algorithmen gelingen. Eine solche Kombination “einfacher” HNF–Algorithmen wird im Rahmen dieser Arbeit als *Hybrid–HNF–Algorithmus* bezeichnet und arbeitet prinzipiell gemäß der folgenden Vorgehensweise:

Die HNF–Berechnung startet mit einem “einfachen” HNF–Algorithmus, der aufgrund der aktuellen Kennwerte der Eingabematrix, d.h. der Eintragsdichte, der Eintragsverteilung, der Eintragsgröße u.a., als geeignet erachtet wird. Während der Durchführung der Berechnung werden die Veränderungen der Kennwerte der (Zwischen-)Matrix protokolliert. Zu dem Zeitpunkt, zu dem ein anderer “einfacher” HNF–Algorithmus aufgrund der veränderten Situation besser geeignet ist, als der bisher verwendete HNF–Algorithmus, paßt man die Verarbeitungsstrategie den veränderten Bedingungen an. Dazu beendet man die Berechnung mit dem aktuellen “einfachen” HNF–Algorithmus und führt die Berechnung auf der zu diesem Zeitpunkt vorliegenden Zwischenmatrix mit dem neuen, besser geeigneten HNF–Algorithmus fort.

Die Untersuchung dieser Idee und ihre Umsetzung ist Gegenstand dieser Arbeit.

Um “einfache” HNF–Algorithmen zu Hybrid–HNF-Algorithmen kombinieren zu können, wurde ein geeignetes C++ Framework entwickelt. Grundlage dieses Frameworks ist zum einen eine Systematik der bekannten “einfachen” HNF–Algorithmen und zum anderen ein geeignetes C++ Klassendesign zur Realisierung von Matrixoperationen. Ein weiterer Bestandteil dieser Arbeit ist eine Arbeitsanweisung, die veranschaulicht, wie das C++ Framework auf eine spezielle Problemstellung hin anzuwenden ist. Dabei wird dargelegt, welche Überlegungen und Schritte notwendig sind, um einen angemessenen Hybrid–HNF–Algorithmus zu konstruieren.

Zum Abschluß dieser Arbeit wird die Leistungsfähigkeit des Frameworks und damit der zugrundeliegenden Idee anhand von Hybrid–HNF–Algorithmen illustriert, die für Matrizen, die im speziellen Umfeld der Klassengruppenberechnung [Jr.99, Nei01] entstehen, und für eine vorgegebene Hardware entwickelt wurden. Umfangreiche Laufzeittabellen illustrieren die Effizienz und somit die Leistungsfähigkeit. Die aktuellen Rekorde im Bereich der Klassengruppen wurden mit Hilfe der in dieser Arbeit entwickleten Hybrid–HNF–Algorithmen berechnet.

Summary

Briefly speaking the Hermite normal form (abbreviated: HNF) of an integer matrix A is an upper triangular matrix H, where the entries above the diagonal are reduced modulo the specific diagonal element: which means, their value is in the range between zero and the specific diagonal element. A matrix A is equivalent to a matrix H if and only if there is an unimodular matrix T with $A \cdot T = H$.

During the last years a lot of algorithms for computing the HNF were developed, starting with the algorithms of Charles Hermite [Her51] to the algorithms of George Havas [HM94a].

In the process of HNF computations with large, integer matrices on a specific machine, one is confronted with the following problem: In the beginning, the density of the non–zero entries and later also the entry size of the intermediate entries are growing in an uncontrollable manner. In general this leads to a very intense use of storage and finally to the abortion of the computation.

In order to solve this problem a number of optimized algorithms were developed mainly for dense matrices [Coh93, Dom89, DKJ87, Hav91]. However for matrices, which are generated in the context of class group computations, these optimizations are not sufficient. I.e., it is not possible to compute the HNF of these matrices with the current hardware available. This is the motivation of this thesis.

The discussion in this thesis is based on the observation, that different "simple" HNF algorithms (known from literature), if applied to the same matrix, act differently at runtime with respect to storage use and development of density. If it is not possible for a single one of the "simple" HNF algorithms to compute the HNF of an integer matrix under the restrictions of the available hardware resources, a useful combination of such algorithms could succeed. In the following such a combination is called *hybrid HNF algorithm* and would function as follows:

The computation is started with a "simple" HNF algorithm, which is chosen based on the actual parameters of the matrix, i.e., density, entry distribution and entry size. During the process of the computation we keep track of the changes of the parameters of the matrix. Whenever a different "simple" HNF algorithm is more suitable than the one used so far (because of the changed parameters) one switches to the new "simple" HNF algorithm.

The analysis of this idea and its implementation are the topic of this thesis.

In order to combine "simple" HNF algorithms to a hybrid HNF algorithm a C++ framework is introduced. For that purpose a system of known "simple" HNF algorithms and a suitable C++ class design for the implementation of matrix operations is developed. In addition to that instructions on how the C++ framework has to be applied to a special problem are given, i.e., it is shown what has to be considered for the construction of an efficient hybrid HNF algorithm.

As conclusion of this thesis the functionality of our framework is demonstrated in the special area of class group computations. The current records in the area of class groups were computed with the help of these hybrid HNF algorithms.

An dieser Stelle möchte ich mich bei all denen bedanken, die mich während der Entstehung dieser Dissertation unterstützt haben.

An erster Stelle danke ich Prof. Dr. Johannes Buchmann, der es mir ermöglicht hat, zusammen mit netten Kollegen an einem interessanten Thema wissenschaftlich zu arbeiten. Durch zahlreiche Diskussionen und durch die kritische Betrachtung der erzielten Resultate verhalf er mir immer wieder zu neuen Einsichten und Erkenntnissen.

Weiterhin danke ich Dr. Ingrid Biehl, Dr. Michael Jacobson, Dr. Markus Maurer, Stefan Neis und Dr. Susanne Wetzel. Die gemeinsame Arbeit an der Entwicklung von LiDIA und die enge Zusammenarbeit auf vielen anderen Gebieten hat mir sehr viel Spaß gemacht und mich wissenschaftlich voran gebrachte. Bei Dr. Ingrid Biehl, Stefan Neis und Dr. Susanne Wetzel möchte mich außerdem ganz herzlich für das Korrekturlesen dieser Arbeit bedanken.

Die Zeit meiner Dissertation war auch eine Zeit vieler tiefgreifender Veränderungen in meinem privaten und beruflichen Leben. Ganz besonders gilt somit mein Dank meiner Ehefrau Anja Theobald, meinen Eltern Marianne und Walter Theobald, meiner Schwester Anja Theobald, sowie meiner Oma Maria Bauer, die mir durch ihren Rückhalt die nötige Energie und Schaffenskraft gaben, um diese Arbeit fertigzustellen.

Abschließend möchte ich mich auch bei meinen Arbeitskollegen der USD GmbH bedanken. Trotz dem Wechsel vom Hochschulleben ins Arbeitsleben räumten sie mir durch ihre Rücksichtnahme die Möglichkeit ein, meine Arbeit zu beenden.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Der Begriff der Hermite–Normalform (HNF) geht auf den französischen Mathematiker C. Hermite zurück, der bereits 1851 in seiner Arbeit *Sur l'indroduction des variables continues dans la theorie des nombres* [Her51] diesen Begriff eingeführt und eine Reihe von Eigenschaften dieser Normalform ganzzahliger Matrizen bewiesen hat.

Die effiziente, computergestützte Berechnung der HNF von dichtbesetzten, ganzzahligen Matrizen ist keine leichte Aufgabe, wie die Vielzahl von Arbeiten zu diesem Thema belegen [Coh93, Dom89, DKJ87, Hav91]. Ebenso stellt die effiziente Berechnung der HNF von großen, dünnbesetzten, ganzzahligen Matrizen eine Herausforderung dar, deren Schwierigkeitsgrad sich nochmals erhöht, wenn die Rahmenbedingungen der zur Verfügung stehende Hardwareplattform berücksichtigt werden. Selbst die gängigen, etablierten Computer–Algebra–Systeme (CAS) wie Pari [BBCO99], Maple [Map99], Mathematica [Mat99] und Magma [Mag99] sind derzeit nicht in der Lage, für dünnbesetzte, ganzzahlige Matrizen mit mehreren tausend Zeilen und Spalten die HNF unter Berücksichtigung der zur Verfügung stehenden Ressourcen, d.h. Hauptspeicher, Auslagerungsspeicher und Rechenzeit, zu berechnen.

In dieser Arbeit stellen wir ein Framework vor, welches den Besonderheiten der Problemstellung, die HNF von großen, dünnbesetzten, ganzzahligen Matrizen zu berechnen, Rechnung trägt und somit uns die Möglichkeit gibt, dieses Problem in Anpassung an die zur Verfügung stehende Hardwareplattform zu bewältigen.

Die weitere Arbeit gliedert sich in fünf Teile:

Im ersten Teil analysieren wir die Problemstellung dieser Arbeit im Detail, führen notwendige Grundlagen ein und schaffen somit die Ausgangsbasis für unsere weiteren Betrachtungen.

Anschließend stellen wir im zweiten Teil die dem Framework zugrundeliegende Idee vor, nämlich verschiedene HNF–Algorithmen zu Hybrid–HNF–Algorithmen zu kombinieren, um auf diese Weise sich während der Durchführung der HNF–Berechnung ändernden Matrixeigenschaften Rechnung zu tragen.

Im dritten Teil dieser Arbeit gehen wir auf die C++ Daten- und Klassenstrukturen ein, die unserer Implementierung von Matrizen in LiDIA zugrunde liegen und einen wesentlichen Teil des in dieser Arbeit dargestellten Frameworks ausmachen. LiDIA ist eine Bibliothek zur Algorithmischen Zahlentheorie, die seit 1994 am Lehrstuhl Prof. Johannes Buchmann entwickelt wird [BBP95, Pap97, Gro99]. Für weitere Details verweisen wir auf die LiDIA–Homepage [Gro99]. In dem gleichen Teil beschreiben wir außerdem die verwendeten Strukturen und Implementierungstechniken. Dabei begründen wir unsere Auswahl, indem wir die Vor- und Nachteile der jeweiligen Strukturen und der jeweiligen Implementierungstechniken gegeneinander abwägen und deren Vor- bzw. Nachteile anhand von einfachen Beispielen herausarbeiten.

Im vierten Teil entwickeln wir eine Systematik für HNF–Algorithmen und sortieren die bekannten Algorithmen zur Berechnung der HNF in diese Systematik ein. Wir erweitern einige der bekannten Algorithmen durch Heuristiken und stellen einige neue Algorithmen zur Berechnung der HNF vor. Desweiteren sammeln wir Laufzeitinformationen, die uns bei der späteren Anwendung unseres Frameworks auf eine spezielle Problemstellung wertvolle Hinweise zur Konstruktion eines Hybrid–HNF–Algorithmus liefern.

Im fünften und letzten Teil beschreiben wir zunächst allgemein, wie man das Framework auf eine spezielle Problemstellung und auf eine spezielle Hardwareplattform anwendet. Anschließend stellen wir die Hybrid–HNF–Algorithmen vor, die wir für Matrizen, die im speziellen Umfeld der Klassengruppenberechnung entstehen, und für eine vorgegebene Hardware gemäß der zuvor beschriebenen Vorgehensweise entwickelt haben. Anhand umfangreicher Laufzeittabellen illustrieren wir die Effizienz und somit die Anwendbarkeit unseres Frameworks. Mit Hilfe der in diesem Abschnitt vorgestellten Hybrid–HNF–Algorithmen wurden die aktuellen Rekorde im Bereich der Klassengruppen berechnet. Diese haben dabei ihre Leistungsfähigkeit unter Beweis gestellt, indem sie die HNF von Matrizen mit mehr als 10.000 Zeilen und Spalten auf einer vorgegebenen Hardwareplattform berechnet haben.

Teil I

Analyse der Problemstellung

Kapitel 2

Grundlagen und Begriffe

In diesem Kapitel führen wir Bezeichnungen, Notationen und Ergebnisse ein, die im weiteren Verlauf der Arbeit benutzt, dort aber nicht mehr explizit erwähnt werden. Als Referenz dienen die Bücher von Lamprecht [Lam78], Cohen [Coh93] und Newman [New72].

Zur Bezeichnung von Zahlenbereichen werden die folgenden Symbole verwendet:

$\mathbb{N}$	Menge der natürlichen Zahlen
$\mathbb{P}$	Menge der Primzahlen
$\mathbb{Z}$	Menge der ganzen Zahlen
$\mathbb{Z}/n\mathbb{Z}$	Restklassenring modulo n
$\mathbb{F}p$	Primkörper mit p Elementen, wobei $p \in \mathbb{P}$

2.1 Vektoren und Matrizen

2.1. Definition (Matrix, Zeilenvektor, Spaltenvektor)
Ein rechteckiges Schema

$$A = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{pmatrix} = (a_{i,j}) \begin{matrix} 0 \le i < m \\ 0 \le j < n \end{matrix}$$

von Elementen aus einer beliebigen Menge M heißt **Matrix** *über M oder genauer eine $(m \times n)$–Matrix, das heißt eine Matrix mit m Zeilen und n Spalten.*

Eine $(m \times 1)$–Matrix

$$A = \begin{pmatrix} a_{0,0} \\ a_{1,0} \\ \vdots \\ a_{m-1,0} \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{pmatrix}$$

heißt **Spaltenvektor**.

Eine $(1 \times n)$*–Matrix*

$$A = \quad (a_{0,0}a_{0,1}\dots a_{0,n-1}) = \quad (a_0a_1\dots a_{n-1})$$

heißt **Zeilenvektor**.

Notationen:

- $a_{*,i}$ mit $i \in \{0,\dots,n-1\}$ bezeichnet die i–te Spalte der Matrix A.
- $a_{j,*}$ mit $j \in \{0,\dots,m-1\}$ bezeichnet die j–te Zeile der Matrix A.

2.2. Bemerkung
Im Hinblick auf die Implementierung beginnen die Numerierungen von Zeilen, Spalten und Elementen bei 0 und nicht, wie sonst üblich, bei 1.

Notationen:

- Zur Vereinfachung bezeichnen wir die Menge aller $(m \times n)$–Matrizen über einer Menge M mit $\mathbf{Mat}_{m\times n}(M)$.
- $I_n \in \mathbf{Mat}_{n\times n}(\mathbb{Z})$ bezeichnet im folgenden die $(n \times n)$–Einheitsmatrix und $e_i^n \in \mathbb{Z}^n$ den i-ten Einheitsvektor aus $\mathbb{Z}^n$.

2.2 Hermite–Normalform

Um zu dem zentralen Begriff dieser Arbeit zu gelangen, müssen wir zunächst definieren, wann eine Matrix zu einer anderen äquivalent ist. Dazu benötigen wir den Begriff der unimodularen Matrix.

2.3. Definition (Unimodulare Matrix)
Eine Matrix $U \in \mathbf{Mat}_{n\times n}(\mathbb{Z})$ *heißt* **unimodular**, *wenn die Determinante von* U *ein Element der Einheitengruppe von* $\mathbb{Z}$, *d.h.* ± 1, *ist.*

Notation:
$\mathrm{GL}_n(\mathbb{Z})$ bezeichnet die Menge aller Matrizen $U \in \mathbf{Mat}_{n\times n}(\mathbb{Z})$ mit $\det(U) = \pm 1$.

Aufbauend auf dem Begriff der unimodularen Matrix können wir eine Äquivalenzrelation auf der Menge $\mathbf{Mat}_{m\times n}(\mathbb{Z})$ wie folgt definieren.

2.4. Definition (Matrixäquivalenz)
Seien $A, B \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$. *Die Matrix* B *heißt* **rechtsäquivalent** *zu* A, *wenn eine unimodulare Matrix* $U \in \mathrm{GL}_n(\mathbb{Z})$ *existiert, so daß gilt:* $B = A \cdot U$ *Analog heißt eine Matrix* B **linksäquivalent** *zu* A, *wenn eine unimodulare Matrix* $U \in \mathrm{GL}_m(\mathbb{Z})$ *mit* $B = U \cdot A$ *existiert.*

2.5. Lemma
Die oben definierten Matrixäquivalenzen sind Äquivalenzrelationen.

Beweis: Siehe [New72]. ■

Wir gelangen somit zum zentralen Begriff dieser Arbeit, dem Begriff der **Hermite–Normalform** (abgekürzt: HNF).

2.6. Definition (Hermite–Normalform)
Eine Matrix $A = (a_{i,j}) \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ ist in **Hermite–Normalform**, *wenn ein $r < n$ und eine streng monoton wachsende Funktion $f : [r, n-1] \to [0, m-1]$ existieren, die den folgenden Eigenschaften genügen:*

1. *Für $r \leq j < n$ gilt:*
   (a) $a_{f(j),j} \geq 1$,
   (b) $a_{i,j} = 0$ *für* $i > f(j)$ *und*
   (c) $0 \leq a_{f(k),j} < a_{f(k),k}$ *für* $k < j$.
2. *Die ersten r Spalten von A sind Nullspalten, d.h. sie bestehen nur aus Nullelementen.*

Diese Definition bedeutet insbesondere, daß eine Matrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ in HNF die folgende Struktur besitzt, wenn $n \geq m$ und $\mathrm{rank}_{\mathbb{Z}}(A) = m$ ist:

$$\begin{pmatrix} 0 & 0 & \cdots & 0 & * & * & \cdots & * \\ 0 & 0 & \cdots & 0 & 0 & * & \cdots & * \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & * \end{pmatrix}$$

2.7. Bemerkung

- Für den wichtigen Spezialfall $m = n$ und $f(k) = k$ ist eine Matrix A in HNF, wenn sie den folgenden Bedingungen genügt:
  1. M ist eine obere Dreiecksmatrix, d.h. $a_{i,j} = 0$ für $i > j$.
  2. Für jedes i gilt: $a_{i,i} > 0$.
  3. Für alle $j > i$ gilt: $0 \leq a_{i,j} < a_{i,i}$.
- In dieser Arbeit wird die Hermite–Normalform ohne Beschränkung der Allgemeinheit als obere Dreiecksmatrix definiert. Die Ergebnisse übertragen sich ohne Einschränkung auf andere Definitionen der Hermite–Normalform (vgl. [Wag97]).

Der folgende Satz ist von zentraler Bedeutung für diese Arbeit. Er bildet die Grundlage für unsere weiteren Betrachtungen.

2.8. Satz (Existenz und Eindeutigkeit der HNF)
Zu jeder Matrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ existiert eine eindeutig bestimmte, zu A rechtsäquivalente Matrix $H \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ in Hermite–Normalform.

Beweis: Zum Beweis dieses Satzes geben wir einen Algorithmus zur Berechnung der HNF an. Dieser Algorithmus basiert auf der Gaußelimination, ist offensichtlich korrekt und berechnet zu einer Eingabematrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ eine Matrix H in HNF mit $H = A \cdot U$,$U \in \mathrm{GL}_n(\mathbb{Z})$.

2.9. Algorithmus

Algorithmus zur Berechnung der HNF: HNF_{Orig}

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$
AUSGABE: $\mathrm{HNF}(A) \in \mathbf{Mat}_{m\times n}(\mathbb{Z}), U \in \mathrm{GL}_n(\mathbb{Z})$ mit $A \cdot U = \mathrm{HNF}(A)$.

[Initialisierung]
(1) $H = A; U = I_n;$
(2) $i = m-1; j = n-1;$
(3) **while** $(i >= 0 \wedge j >= 0)$ **do**
[Elimination]
(4) $v = h_{i,*};$
(5) **if** $((h_{i,0}, \ldots, h_{i,j}) = (0, \ldots, 0))$ **then**
(6) $i--;$
(7) **else**
(8) **for** $(l = 1; l < j; l++)$ **do**
(9) **if** $(v_{l-1} \neq 0)$ **then**
(10) **if** $(v_l \neq 0)$ **then**
(11) g = xgcd(v_{l-1},v_l, a, b); // $g = a \cdot v_{l-1} + b \cdot v_l$
(12) $\hat{U} = \begin{pmatrix} I_{l-1} & & & \\ & -\frac{v_l}{g} & a & \\ & \frac{v_{l-1}}{g} & b & \\ & & & I_{n-l} \end{pmatrix};$
(13) $H = H \cdot \hat{U};$
(14) $U = U \cdot \hat{U};$
(15) **else**
(16) $A.swap\_columns(l-1, l);$ // $a_{*,l-1} \longleftrightarrow a_{*,l}$
(17) $U.swap\_columns(l-1, l);$ // $u_{*,l-1} \longleftrightarrow u_{*,l}$
(18) **fi**
(19) **fi**
(20) **od**
[Normalisierung]
(21) **for** $(l = j+1; l < n-1; l++)$ **do**
(22) $pos\_div\_rem(q, r, v_l, v_j)$
(23) $h_{*,l} = h_{*,l} - q \cdot h_{*,j};$

```
(24)            u_{*,l} = u_{*,l} − q · u_{*,j};
(25)          od
(26)          i − −; j − −;
(27)        fi
(28)      od
(29)    od
(30)    return (H,U);
```

2.10. Bemerkung [LiDIA]

- Die Funktion $xgcd(a, b, c, d)$ berechnet den größten gemeinsamen Teiler g der Zahlen a und b inklusive Darstellung, gibt den größten gemeinsamen Teiler g als Ergebnis zurück und speichert die Koeffizienten der Darstellung in den Variablen c und d, so daß gilt: $a \cdot c + b \cdot d = g$
- Die Funktion $pos\_div\_rem(q, r, a, b)$ dividiert a mit Rest durch b, so daß gilt $a = b \cdot q + r$, wobei r der kleinsten, nicht negativen Zahl entspricht, die diese Gleichung erfüllt.
- Die Funktion $A.swap\_columns(i, j)$ vertauscht die Spalten i und j der Matrix A.

Der obige Algorithmus transformiert zeilenweise die Eingabematrix A in eine obere Dreiecksmatrix und reduziert dabei die Elemente der jeweiligen Zeile modulo dem aktuellen Diagonalelement und stellt somit die HNF–Bedingung sicher.

Es ist offensichtlich, daß die Ergebnismatrizen H und U des obigen Algorithmus das Gewünschte leisten.

Die Beweis der Eindeutigkeit findet sich z.B. in [New72].

■

2.11. Bemerkung
Eine wichtige Beobachtung ist, daß die unimodulare Transformationsmatrix U für Matrizen $A \in \mathbf{Mat}_{n\times n}(\mathbb{Z})$ mit $\text{rank}_{\mathbb{Z}}(A) = n$ eindeutig bestimmt ist. Denn es gilt:

$$U \quad = \quad \frac{1}{\det(A)} \,\text{adj}(A) \cdot \text{HNF}(A)$$

2.3 Untersuchungskriterien, Normen und Schranken

In diesem Abschnitt definieren wir Kriterien, mit deren Hilfe wir im weiteren Verlauf dieser Arbeit die Qualität von Algorithmen zur Berechnung der Hermite–Normalform beurteilen

werden. Desweiteren definieren wir Normen und Schranken, die wir zur Beschreibung von Algorithmen und Algorithmenparametern benötigen werden.

Wie bereits dargelegt, beschäftigen wir uns in dieser Arbeit mit der Normalformberechnung von großen, dünnbesetzten, ganzzahligen Matrizen. Somit spielt die Anzahl der von Null verschiedenen Einträge eine besondere Rolle.

2.12. Definition (Gewicht einer Matrix, Eintragsdichte einer Matrix)
Das **Gewicht** ω *einer Matrix* $A = (a_{i,j}) \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ *wird als die Anzahl der von Null verschiedenen Einträge in* A *definiert, also*

$$L_0(A) := \omega(A) \quad := \quad |\{a_{i,j} | a_{i,j} \neq 0, 0 \leq i < m, 0 \leq j < n\}|.$$

Die **Eintragsdichte** ϱ *einer Matrix* $A = (a_{i,j}) \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ *wird als*

$$\varrho(A) \quad := \quad \frac{\omega(A) \cdot 100}{m \cdot n}$$

definiert.

Diese Definition gilt analog für Vektoren.

2.13. Definition (Gewicht eines Vektors, Eintragsdichte eines Vektors)
Das Gewicht eines Vektors $v \in \mathbb{Z}^n$ *ist definiert als*

$$L_0(v) := \omega(v) := |\{v_i | v_i \neq 0, 0 \leq i < n\}|.$$

Die **Eintragsdichte** ϱ *eines Vektors* $v \in \mathbb{Z}^n$ *ist definiert als:*

$$\varrho(v) \quad := \quad \frac{\omega(v) \cdot 100}{n}$$

Desweiteren zeigt die Praxis, daß die Verteilung der Nicht–Nulleinträge einer Matrix A Einfluß auf das Laufzeitverhalten der betrachteten Algorithmen hat. Wir benötigen somit die folgenden Begriffe:

2.14. Definition (Spaltenverteilungsfunktion)
Unter der **Spaltenverteilungsfunktion** *einer* $(m \times n)$*–Matrix* A *versteht man die Abbildung* $v_s(A) : [0, n-1] \to [0, m]$*, die jedem Index einer Spalte die Anzahl der Nicht–Nullelemente der jeweiligen Spalte einer Matrix* A *zuordnet.*

2.15. Definition (Zeilenverteilungsfunktion)
Unter der **Zeilenverteilungsfunktion** *einer* $(m \times n)$*–Matrix* A *versteht man die Abbildung* $v_z(A) : [0, m-1] \to [0, n]$*, die jedem Index einer Zeile die Anzahl der Nicht–Nullelemente der jeweiligen Zeile einer Matrix* A *zuordnet.*

Neben der Anzahl und der Verteilung der Einträge ungleich Null ist die Größe der Einträge für den benötigten Speicherplatz und somit implizit für die Verarbeitungsgeschwindigkeit von Bedeutung. Aus diesem Grund benötigen wir die folgenden Begriffe.

Für Matrizen definieren wir die folgende Norm:

2.16. Definition (Maximumsnorm einer Matrix)
Die Maximumsnorm einer $(m \times n)$–Matrix $A = (a_{i,j}) \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ ist definiert als

$$L_\infty(A) \quad := \|A\| := \quad \max\{|a_{i,j}| \ | 0 \leq i < m, 0 \leq j < n\}.$$

2.17. Definition (durchschnittliche Eintragsgröße der Einträge einer Matrix)
Die durchschnittliche Eintragsgröße Δ_∞ einer Matrix $A = (a_{i,j}) \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ ist definiert als

$$\Delta_\infty(A) := \frac{1}{\max(L_0(A), 1)} \cdot \sum_{\substack{a_{i,j} \neq 0 \\ 0 \leq i < m \\ 0 \leq j < n}} (|a_{i,j}|)$$

Auf der Ebene der Vektoren benötigen wir die folgenden Begriffe:

2.18. Definition (Maximumsnorm eines Vektors)
Die Maximumsnorm eines Vektors $v \in \mathbb{Z}^n$ ist definiert als

$$L_\infty(v) \quad := \|v\| := \quad \max\{|v_i| \ | 0 \leq i < n\}.$$

2.19. Definition (k–Norm)
Die k–Norm eines Vektors $v \in \mathbb{Z}^n$ mit $k \in \mathbb{N}$ ist definiert als

$$L_k(v) := \left(\sum_{i=0}^{n-1} (|v_i|^k) \right)^{\frac{1}{k}}.$$

Zum Abschluß definieren wir die folgende Schranke, die wir zur Abschätzung der Größe von (Zwischen-)Ergebnissen benötigen.

2.20. Definition (Hadamard–Schranke)
Die Hadamard–Schranke einer Matrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ ist definiert durch:

$$\text{Hadamard}(A) \quad := \quad \min\{ \prod_{j=0}^{n-1} L_2(a_{*,j}), \prod_{j=0}^{m-1} L_2(a_{j,*}) \}$$

2.4 Modulare Arithmetik

Der Begriff *modulare Arithmetik* bezeichnet im weiteren Verlauf dieser Arbeit eine Vorgehensweise, die es ermöglicht, die Berechnung eines *eindeutig* definierten Objektes über $\mathbb{Z}$ auf die Berechnung kongruenter, eindeutig definierter Objekte in mehreren endlichen Körpern K_i zu reduzieren und somit die Größe der entstehenden Zwischeneinträge zu kontrollieren.

Die Grundlage dieser Vorgehensweise bildet der folgende Satz.

2.21. Satz (Chinesischer Restsatz)
Es seien $m_1, \ldots, m_n \in \mathbb{N}$ *paarweise teilerfremd und es seien* $x_1, \ldots, x_n \in \mathbb{Z}$. *Dann existiert ein* $x \in \mathbb{Z}$ *mit:*

$$x \equiv x_i \bmod m_i \qquad \textit{für } 1 \leq i \leq n$$

Dieses x *ist eindeutig bestimmt modulo* $S = \prod_{i=1}^{n} m_i$.

Beweis: Siehe [Coh93]. ■

Wählen wir die Parameter $m_1, \ldots, m_n$ aus der Menge aller Primzahlen ($\mathbb{P}$) und S hinreichend groß, bietet uns dieser Satz die Möglichkeit, die Berechnung eines *eindeutig* definierten Objektes *Obj* über $\mathbb{Z}$ auf die Berechnung kongruenter, eindeutig definierter Objekte Obj_i in mehreren endlichen Körpern $K_i = \mathbb{F}_{m_i},\ i = 1, \ldots, n$ zu reduzieren. Die Berechnung dieses Objektes bzw. die Lösung eines Problems in $\mathbb{Z}$ erfolgt dann nach dem in der folgenden Abbildung dargestellten Schema.

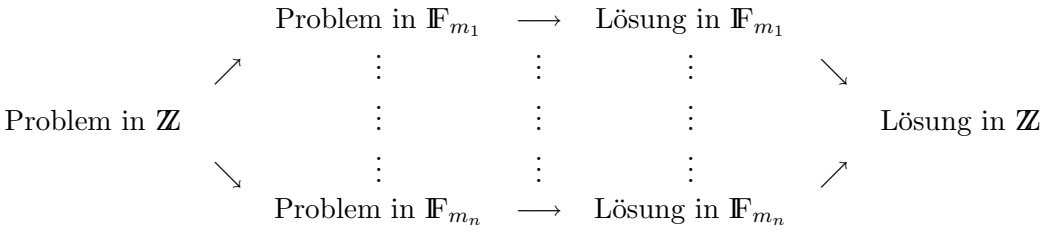


Abbildung 2.1: Schema der modularen Arithmetik

Für die praktische Anwendung dieser Vorgehensweise stellt sich die Frage, welche Berechnungen über $\mathbb{Z}$ man auf die beschriebene Art und Weise auf Berechnungen über endlichen Körpern verlagern kann.

Aus dem Chinesischen Restsatz erhalten wir implizit zwei Kriterien, die uns bei der Beantwortung dieser Fragestellung helfen.

1. Damit eine Berechnung auf endliche Körper verlagert werden kann, muß die Lösung der Berechnung über $\mathbb{Z}$ modulo m_i kongruent zu den Lösungen adäquater, eindeutig definierter Berechnungen über den endlichen Körpern $\mathbb{F}_{m_i}$ sein.

2. Desweiteren müssen wir für die Komponenten der Lösung über $\mathbb{Z}$ eine obere Schranke S angeben oder berechnen können.

Die in der Tabelle 2.1 aufgeführten Berechnungen in Verbindung mit der jeweils angegebenen Schranke erfüllen offensichtlich diese beiden Kriterien.

Berechnung	**Schranke**
Determinantenberechnung einer Matrix $A \in \mathbf{Mat}_{n\times n}(\mathbb{Z})$	Hadamard(A)
Berechnung der adjungierten Matrix zu $A \in \mathbf{Mat}_{n\times n}(\mathbb{Z})$	Hadamard(A)
Berechnung des Rangs einer Matrix $B \in \mathbf{Mat}_{n\times m}(\mathbb{Z})$	Hadamard(B)
Berechnung des Rangs und der Indizes der linear unabhängigen Zeilen einer Matrix $B \in \mathbf{Mat}_{n\times m}(\mathbb{Z})$	Hadamard(B)
Berechnung des Rangs und der Indizes der linear unabhängigen Spalten einer Matrix $B \in \mathbf{Mat}_{n\times m}(\mathbb{Z})$	Hadamard(B)
Berechnung der Koeffizienten des charakteristischen Polynoms einer Matrix $A \in \mathbf{Mat}_{n\times n}(\mathbb{Z})$	$n \cdot \|A\|^n$

Tabelle 2.1: Problemstellungen und Schranken der modularen Arithmetik

Betrachtet man die Vorgehensweise der modularen Arithmetik aus dem Blickwinkel einer geeigneten Modularisierung, erhalten wir eine Dreiteilung des Problemlösevorgangs. Sei A eine ganzzahlige Matrix.

Phase 1: Vorberechnung
Zunächst berechnen wir zwecks Abschätzung der Größe der Komponenten des erwarteten Resultats Obj eine obere Schranke S. Anschließend ermitteln wir eine Liste von Primzahlen $m_1, \ldots, m_n$, deren Produkt größer als das Doppelte der berechneten Schranke ist. Damit haben wir die Voraussetzungen geschaffen, um mittels des Chinesischen Restsatzes die richtige Lösung in $\mathbb{Z}$ zu finden.

Phase 2: Reduktion
Die in Phase 1 ermittelten Primzahlen dienen uns als Charakteristiken m_i der Primkörpern $\mathbb{F}_{m_i}$, auf die unsere Problemstellung jetzt übertragen wird. Anschließend wird das Problem innerhalb der Primkörper der jeweiligen Charakteristik gelöst.

Phase 3: Rekonstruktion
Die Ergebnisse $Obj_1, \ldots, Obj_n$ der Berechnungen aus Phase 2 werden gesammelt und in Phase 3 mittels des oben angegebenen Chinesischen Restsatzes zu einer Lösung in $\mathbb{Z}$ zusammengesetzt.

Dementsprechend benötigen wir zur Realisierung dieses Ansatzes folgende Funktionen:

1. **hadamard**: Diese Funktion erhält als Eingabeparameter eine ganzzahlige Matrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ und berechnet eine Approximation der Hadamard–Schranke.

2. **get_primlist**: Diese Funktion erhält als Eingabeparameter eine Schranke S und liefert als Ergebnis eine Primzahlliste $p_1, \ldots, p_n$ zurück mit $\prod_{i=1}^{n} p_i > 2 \cdot S$.

3. **chinrest**: Diese Funktion existiert in verschiedenen Ausprägungen, die sich lediglich in dem Objekt unterscheiden, das mittels des Chinesischen Restsatzes zusammengesetzt wird. Eine Ausprägung setzt aus Matrizen über endlichen Körpern eine Matrix über $\mathbb{Z}$, eine andere aus Vektoren über endlichen Körpern einen Vektor über $\mathbb{Z}$ und eine weitere einzelne Elemente aus endlichen Körpern zu einem Element aus $\mathbb{Z}$ zusammen.

Auf weitere Details dieser Funktionen und dieses Ansatzes gehen wir an dieser Stelle nicht ein, sondern verweisen auf die detaillierten Untersuchungen in [Mül94] und [The95]. Die dort ermittelten Ergebnisse bzgl. der Verwendung von long– und bigint–Implementierungen und Optimierungen übertragen sich analog auf Implementierungen, die sich dieses Ansatzes bedienen.

2.22. Bemerkung [LiDIA]
Die LiDIA–Klasse *bigint* repräsentiert $\mathbb{Z}$ und bietet mittels einer Multi–Precision–Arithmetik Operation wie Addition, Multiplikation, Moduloreduktion, Berechnung des größten gemeinsamen Teilers inklusive Darstellung usw. an.

2.5 Primzahlgenerierung

Im vorangegangenen Abschnitt bestand eine der Aufgaben darin, eine Liste von Primzahlen zu ermitteln, deren Produkt das Doppelte einer vorgegebenen Schranke überschreitet. In diesem Abschnitt widmen wir uns der Aufgabe, eine Primzahl einer vorgegebenen Bitlänge zu finden.

Dazu bedienen wir uns der folgenden Vorgehensweise:

1. Wähle eine Zufallszahl in der gewünschten Bitlänge.
2. Prüfe z.B. mit dem Algorithmus von Miller–Rabin [BS96], ob diese Zahl eine Primzahl ist. Ist dies nicht der Fall, beginne wieder bei Schritt 1.

Dies alles führt zu dem Algorithmus **compute_prime**, der als Eingabe eine ganze Zahl n erhält und eine Zahl berechnet, die mit hoher Wahrscheinlichkeit eine Primzahl ist. Weiterführende Informationen finden sich in [BS96] und [CLR89].

2.6 Gitter

Einige Algorithmen zur Berechnung der HNF machen sich Eigenschaften des zu einer Matrix A gehörenden Gitters $L(A)$ zunutze. Zu deren Beschreibung werden folgende Definitionen und Zusammenhänge benötigt [Wet98, New72].

2.23. Definition (Gitter im $\mathbb{Z}^m$)
Ein **Gitter** *L in $\mathbb{Z}^m$ ist eine additive Untergruppe des $\mathbb{Z}^m$, so daß*

$$L = \left\{ \sum_{i=0}^{n-1} x_i \cdot b_i | x_i \in \mathbb{Z}, i = 0, \ldots, n-1 \right\},$$

wobei die Vektoren $b_0, \ldots, b_{n-1} \in \mathbb{Z}^m$, $n \leq m$ linear unabhängig sind. Wir nennen dann die Matrix $B = (b_0, \ldots, b_{n-1}) \in \mathbf{Mat}_{m \times n}(\mathbb{Z})$ eine Basis des Gitters $L(B)$ mit der Dimension n. Für $m = n$ heißt das Gitter **volldimensional**.

Es ist eine wohlbekannte und leicht zu verifizierende Tatsache, daß folgende elementare Operationen das von den Spalten einer Matrix $A \in \mathbf{Mat}_{m \times n}(\mathbb{Z})$ erzeugte Gitter $L(A)$ nicht ändern:

2.24. Definition (Elementare Spaltenoperationen)
Als elementare Spaltenoperationen auf einer Matrix $A \in \mathbf{Mat}_{m \times n}(\mathbb{Z})$ werden

- *der Tausch zweier Spalten,*
- *die Addition eines ganzzahligen Vielfachen einer Spalte zu einer anderen Spalte sowie*
- *die Multiplikation einer Spalte mit einer Einheit, d.h. ± 1*

bezeichnet.

Aus der Existenz und der Eindeutigkeit der Hermite–Normalform folgt, daß jedes Gitter in $\mathbb{Z}^m$ eine eindeutig bestimmte Basis in HNF besitzt, was zu dem folgenden Satz führt.

2.25. Satz
Sei $B = (b_0, \ldots, b_{n-1}) \in \mathbf{Mat}_{m \times n}(\mathbb{Z})$ eine Basis des Gitters $L \subset \mathbb{Z}^m$. Dann gilt: $B' = (b'_0, \ldots, b'_{n-1}) \in \mathbf{Mat}_{m \times n}(\mathbb{Z})$ ist ebenfalls eine Basis des Gitters L genau dann, wenn eine unimodulare Matrix $U \in \mathrm{GL}_n(\mathbb{Z})$ existiert, so daß $B \cdot U = B'$.

Beweis: Siehe [New72]. ■

Letztendlich wird noch der Begriff der Gitterdeterminanten benötigt, der für die modularen Algorithmen zur Berechnung der HNF von zentraler Bedeutung ist.

2.26. Definition (Gitterdeterminante)
Die Determinante $\det(L)$ *eines Gitters $L \subset \mathbb{Z}^m$ mit der Basis $B \in \mathbf{Mat}_{m \times n}(\mathbb{Z})$ ist definiert als*

$$\det(L) = \sqrt{|\det(B^T \cdot B)|}$$

Diese Definition ist sinnvoll, da nach Satz 2.25 verschiedene Basen eines Gitters sich mittels unimodularer Matrizen ineinander überführen lassen.

Der folgende Satz liefert uns eine obere Schranke für die Größe der Determinante eines Gitters.

2.27. Satz
Für ein Gitter $L \subset \mathbb{Z}^m$ mit einer Basis $B = (b_0, \ldots, b_{n-1}) \in \mathbf{Mat}_{m \times n}(\mathbb{Z})$ gilt:

$$\det(L) \leq \prod_{i=0}^{n-1} \|b_i\|$$

Beweis: Siehe [DKJ87]. ■

Damit ist leicht zu sehen, daß für eine ganzzahlige $(n \times n)$–Matrix A gemäß dieses Satzes gilt:

$$|det(A)| \leq n^{\frac{n}{2}} \cdot d_0 \cdot d_1 \ldots d_{n-1} \leq (n^{\frac{1}{2}} \|A\|)^n,$$

wobei d_j eine obere Schranke für die Größe der Einträge in der j-ten Spalte von A ist.

Weiterhin gilt die in folgendem Satz beschriebene Eigenschaft.

2.28. Satz
Sei A eine ganzzahlige $(m \times n)$–Matrix, $m \leq n$, und sei d die Gitterdeterminante von $L(A)$. Dann gilt $\forall i \in \{0, \ldots, n-1\}$:

$$d \cdot e_i^m \in L(A)$$

Beweis: Siehe [DKJ87]. ■

2.7 Probabilistische Algorithmen

Probabilistische Algorithmen spielen im weiteren Verlauf dieser Arbeit noch eine Rolle. Wir unterscheiden zwischen einem probabilistischen Algorithmus vom Typ *Las Vegas*, d.h. einem Algorithmus, der mit einer Wahrscheinlichkeit von mindestens 1/2 das korrekte Ergebnis liefert und ansonsten mit einer Fehlermeldung endet, und einem probabilistischen Algorithmus vom Typ *Monte Carlo*, d.h. einem Algorithmus, der mit einer Wahrscheinlichkeit von mindestens 1/2 ein korrektes Ergebnis liefert aber auch ein falsches Ergebnis liefern kann.

Um für einen probabilistischer Algorithmus vom Typ *Las Vegas* zu einer Erfolgswahrscheinlichkeit von $1-\epsilon$ mit $\epsilon \leq 1/2$ zu gelangen, wird der jeweilige Algorithmus $\lceil \log_2(\frac{1}{\epsilon}) \rceil$–mal aufgerufen.

Für einen *Monte Carlo*–Algorithmus zur Lösung eines Problems mit genau einer korrekten Lösung kann die Erfolgswahrscheinlichkeit erhöht werden, indem der Algorithmus mehrfach parallel gestartet wird und als Ergebnis dasjenige ausgewählt wird, welches von der Mehrzahl der Prozesse produziert wird [BBT99].

Kapitel 3

Problemanalyse

Der algorithmische Beweis der Existenz und der Eindeutigkeit der HNF verleitet schnell zu der Annahme, daß die praktische Berechnung der HNF kein Problem darstellt. Leider täuscht diese Annahme. In diesem Kapitel zeigen wir auf, worin das eigentliche Problem besteht, praktisch die HNF von großen, dünnbesetzten, ganzzahligen Matrizen zu berechnen.

Dazu wählen wir uns eine zufällige, dünnbesetzte, ganzzahlige Beispielmatrix aus. Anschließend berechnen wir mittels des im Beweis zur Existenz und Eindeutigkeit der HNF angegebenen Algorithmus die HNF und protokollieren nach jeder durchgeführten Zeilenelimination die Eintragsdichte und die maximale Eintragsgröße der entstandenen Zwischenmatrix. Als Plattform unserer Untersuchung dient uns ein Pentium II 400 MHz mit 128 MB Hauptspeicher und 128 MB Swap (Betriebssystem Linux). Bei einem Hauptspeicherverbrauch von mehr als 128 MB bzw. einer Laufzeit von mehr als 24 Stunden beenden wir die Berechnung vorzeitig.

Als Beispielmatrix haben wir eine (400×800)–Matrix mit einer Eintragsdichte von 2% und zufälligen Einträgen aus dem Intervall $]-100; 100[$ ausgewählt. Die ermittelten Ergebnisse sind in den Abbildungen 3.1 und 3.2 dargestellt. Abbildung 3.1 setzt die Eintragsdichte der entstandenen Zwischenmatrix in Beziehung zur durchgeführten Zeilenelimination. In Abbildung 3.2 wird die Binärlänge des maximalen Eintrags einer Zwischenmatrix in Beziehung zur bearbeiteten Zeile gestellt.

Diese beiden Abbildungen illustrieren deutlich die Probleme, die bei der praktischen Berechnung der HNF auftauchen. Von Zeile zu Zeile wächst die Eintragsdichte der verbleibenden Restmatrix. Ist eine Eintragsdichte von ca. 18% erreicht, wächst zusätzlich von Zeile zu Zeile die Größe der entstehenden Matrixeinträge. Beide Entwicklungen kombiniert führen zu einem explosionsartigen Ansteigen des Speicherplatzbedarfs und schließlich zu Abbruch der Berechnung.

Wir haben es somit mit den folgenden beiden Problemfeldern zu tun:

1. zunehmende Eintragsdichte und
2. zunehmende Größe der Zwischeneinträge

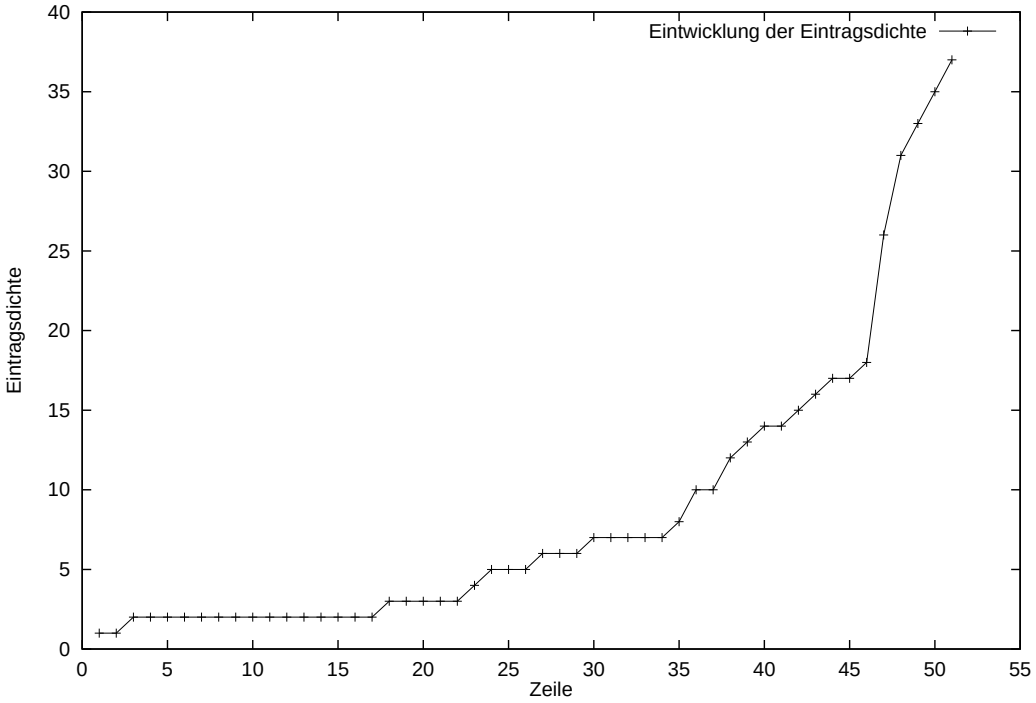


Abbildung 3.1: Entwicklung der Eintragsdichte

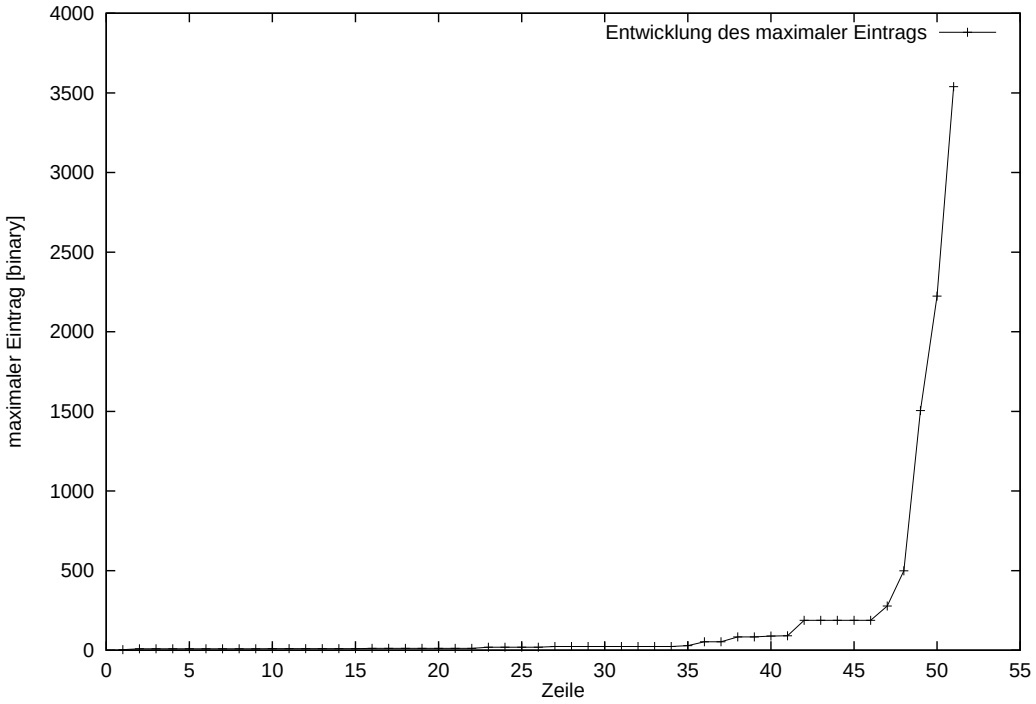


Abbildung 3.2: Entwicklung des maximalen Eintrags

In den folgenden Abschnitten schauen wir dies noch etwas genauer an.

3.1. Bemerkung
Die Computer–Algebra–Systeme Magma 2.2, Mathematica 3.0, Pari 2.0 und Maple 5.4 sind nicht in der Lage, die HNF dieser Beispielmatrix zu berechnen. Speicherplatzmangel bzw. Überschreitung der zur Verfügung stehenden Rechenzeit führten zum Abbruch der Berechnung.

3.1 Zunehmende Eintragsdichte

Widmen wir uns zunächst dem Problem der steigenden Eintragsdichte. Unsere Berechnung startet mit einer dünnbesetzten, ganzzahligen Matrix. Wie wir gesehen haben, nimmt die Eintragsdichte von Zeilenelimination zu Zeilenelimination stetig zu.

Vor dem Hintergrund ressourcenschonende und effiziente Algorithmen konstruieren zu wollen, stellt sich somit die Frage, welche Parameter diese Entwicklung der Eintragsdichte positiv oder negativ beeinflussen.

Offensichtlich spielt die Eintragsdichte zu Beginn der Berechnung eine Rolle. Desweiteren ist es ebenso klar, daß die Anfangsverteilung der Einträge innerhalb der Matrix die Entwicklung der Eintragsdichte beeinflußt. Konzentrieren sich die Einträge zu einem hohen Prozentanteil auf wenige Zeilen der Matrix, kanalisiert sich aufgrund dessen, daß der verwendete Algorithmus zeilenweise arbeitet und Elemente links der aktuellen Arbeitsposition durch Bildung geeigneter Linearkombinationen eliminiert werden, das Anwachsen der Eintragsdicht ebenfalls auf diese wenigen Zeilen. Somit steigt die Eintragsdichte langsamer im Vergleich zu einer Matrix mit gleicher Eintragsdichte und einer Gleichverteilung der Einträge an. Folglich beginnt die explosionsartige Zunahme der Eintragsdichte auch zu einem späteren Zeitpunkt.

Zielsetzung:
Ein Ziel bei der Entwicklung von Algorithmen zur Berechnung der HNF von großen, dünnbesetzten Matrizen muß sein, die Eintragsdichte so lange wie möglich klein zu halten. Das Entstehen neuer Einträge muß auf Bereiche kanalisiert werden, die sich bereits zu Beginn durch eine hohe Eintragsdichte auszeichnen.

3.2 Zunehmende Größe der Einträge

Das zweite Problem, mit dem wir es zu tun haben, ist die zunehmende Größe der Zwischeneinträge und der damit verbundene Speicherplatzbedarf. Das obige Beispiel zeigt, daß der Originalalgorithmus zur Berechnung der HNF zunächst die Dichte der Matrix erhöht, während die Größe der Einträge zunächst konstant bleibt. Ab einer hinreichend großen Eintragsdichte der Restmatrix explodiert die Größe der Einträge.

3.2. Bemerkung
Das Phänomen, daß während einer Berechnung Zwischeneinträge in unkontrollierter Art und Weise wachsen, wird in der Literatur als *intermediate expression swell*, eingeführt von McClellan [McC73], oder *entry explosion*, eingeführt durch Havas und Sterling [HS79], bezeichnet und tritt in vielen Algorithmen der linearen Algebra über $\mathbb{Z}$ auf.

Um die oben beschriebene Beobachtung zu verdeutlichen, wollen wir an dieser Stelle ein weiteres Beispiel anführen. Nehmen wir an, wir hätten eine Matrix $R1$ bereits soweit reduziert, daß die folgende (20×20)–Matrix mit Einträgen kleiner als 10 übrig bleibt. (Diese Matrix ist der Arbeit von Hafner und McCurley [HM91] entnommen.)

$$\begin{pmatrix}
6 & 3 & 1 & 2 & 5 & 4 & 1 & 8 & 6 & 5 & 8 & 5 & 5 & 5 & 9 & 0 & 1 & 2 & 4 & 7 \\
9 & 6 & 3 & 9 & 4 & 4 & 7 & 0 & 7 & 9 & 1 & 9 & 4 & 4 & 4 & 2 & 9 & 4 & 9 & 0 \\
2 & 6 & 7 & 6 & 4 & 2 & 3 & 3 & 8 & 9 & 9 & 4 & 4 & 1 & 8 & 4 & 9 & 1 & 9 & 0 \\
8 & 0 & 5 & 5 & 6 & 4 & 4 & 1 & 4 & 8 & 8 & 6 & 9 & 8 & 0 & 3 & 2 & 5 & 8 & 5 \\
3 & 6 & 3 & 8 & 4 & 6 & 7 & 3 & 2 & 2 & 4 & 0 & 0 & 8 & 7 & 0 & 0 & 8 & 2 & 7 \\
0 & 4 & 3 & 7 & 6 & 0 & 0 & 7 & 9 & 6 & 7 & 6 & 4 & 5 & 2 & 3 & 3 & 2 & 9 & 3 \\
5 & 8 & 4 & 0 & 9 & 7 & 0 & 9 & 0 & 8 & 7 & 6 & 3 & 9 & 8 & 6 & 2 & 6 & 9 & 2 \\
7 & 0 & 1 & 9 & 7 & 9 & 5 & 3 & 3 & 5 & 1 & 8 & 7 & 3 & 6 & 9 & 9 & 7 & 6 & 6 \\
5 & 4 & 8 & 7 & 0 & 6 & 7 & 8 & 8 & 6 & 7 & 2 & 5 & 2 & 7 & 0 & 2 & 4 & 0 & 3 \\
3 & 3 & 3 & 0 & 2 & 5 & 1 & 3 & 6 & 6 & 7 & 1 & 7 & 4 & 1 & 3 & 8 & 1 & 1 & 2 \\
4 & 5 & 4 & 4 & 6 & 1 & 9 & 1 & 9 & 3 & 5 & 3 & 8 & 7 & 9 & 4 & 9 & 9 & 2 & 9 \\
8 & 8 & 4 & 9 & 2 & 2 & 6 & 1 & 3 & 5 & 3 & 3 & 5 & 2 & 2 & 3 & 0 & 9 & 2 & 7 \\
0 & 4 & 7 & 8 & 1 & 5 & 8 & 6 & 7 & 0 & 4 & 5 & 5 & 8 & 2 & 9 & 6 & 2 & 9 & 8 \\
1 & 1 & 6 & 9 & 0 & 2 & 7 & 7 & 7 & 7 & 4 & 7 & 9 & 8 & 9 & 9 & 6 & 5 & 4 & 5 \\
2 & 6 & 0 & 6 & 7 & 2 & 9 & 1 & 6 & 3 & 4 & 3 & 9 & 0 & 7 & 1 & 6 & 2 & 3 & 1 \\
8 & 1 & 4 & 3 & 1 & 5 & 5 & 3 & 0 & 1 & 0 & 4 & 3 & 2 & 0 & 0 & 8 & 2 & 3 & 8 \\
0 & 4 & 3 & 3 & 5 & 9 & 8 & 0 & 3 & 2 & 4 & 3 & 1 & 1 & 9 & 1 & 9 & 6 & 0 & 4 \\
5 & 5 & 0 & 1 & 1 & 6 & 0 & 4 & 1 & 0 & 0 & 4 & 4 & 0 & 2 & 4 & 6 & 9 & 8 & 7 \\
2 & 3 & 1 & 1 & 2 & 1 & 0 & 2 & 9 & 5 & 4 & 3 & 8 & 1 & 7 & 1 & 0 & 1 & 9 & 0 \\
8 & 2 & 7 & 1 & 4 & 4 & 5 & 3 & 8 & 5 & 6 & 9 & 9 & 2 & 6 & 8 & 5 & 5 & 8 & 3
\end{pmatrix}$$

Abbildung 3.3: (20×20)– Beispielmatrix: R1

Diese Matrix wird nun mittels des im Beweis verwendeten Algorithmus in HNF transformiert. Während dieser Berechnung tritt ein interessantes Phänomen auf. Als maximale Zwischenergebnisse erhalten wir Zahlen der Größenordnung 10^{720}.

Obwohl die Einträge der Startmatrix von moderater Größe sind, wachsen die Zwischenergebnisse in unkontrollierter Art und Weise an. In Tabelle 3.1 haben wir die maximalen Werte der Zwischenmatrizen eingetragen, die jeweils nach einer Zeilenelimination entstehen.

Man erkennt, daß bereits nach wenigen Eliminationsschritten der Wertebereich einer *long*–Variablen verlassen wird. Der Gebrauch einer Multi–Precision–Arithmetik scheint unumgänglich zu sein. Es ist einleuchtend, daß das Phänomen der *entry explosion* unangenehme Auswirkungen auf den Speicherplatzbedarf und auch auf die Laufzeit hat.

3.3. Bemerkung
Frumkin führt in seiner Arbeit [Fru77] aus, daß für bestimmte Matrizen die Einträge als Ergebnis einer Spaltenoperation in obigem Sinne quadriert werden können, obwohl bisher

Eliminationsschritt	maximaler Wert
1	78
2	189
3	418
4	252404
5	7576858302412
6	508959989329979740
7	1143378883295124024912426
8	$\approx 10^{54}$
9	$\approx 10^{63}$
10	$\approx 10^{73}$
11	$\approx 10^{83}$
12	$\approx 10^{93}$
13	$\approx 10^{106}$
14	$\approx 10^{324}$
15	$\approx 10^{338}$
16	$\approx 10^{690}$
17	$\approx 10^{706}$
18	$\approx 10^{720}$
19	$\approx 10^{354}$
20	$\approx 10^{354}$

Tabelle 3.1: Entwicklung des maximalen Eintrags

keine solche Matrix bekannt ist. Dies legt die Möglichkeit nahe, daß ein gegebener Eintrag einer Matrix $A \in \mathbf{Mat}_{n\times n}(\mathbb{Z})$, sagen wir $a_{i,j}$, zu einer Größe von $|a_{i,j}|^{2^n}$ anwachsen kann.

Zielsetzung:
Ein Ziel bei der Entwicklung von Algorithmen zur Berechnung der HNF von großen, dünnbesetzten, ganzzahligen Matrizen muß also sein, Mechanismen zu Verhinderung bzw. zur Eindämmung der *entry explosion* zu untersuchen und zu integrieren.

Teil II

Lösungsidee: Ein Framework zur Konstruktion von Hybrid–HNF–Algorithmen

Kapitel 4

Idee: Kombination von HNF–Algorithmen

Seit der Entdeckung der HNF durch C. Hermite bis zum heutigen Tag wurden eine Vielzahl unterschiedlicher Algorithmen zur Berechnung der HNF entwickelt, getestet und in dem jeweiligen Umfeld, für das sie entwickelt wurden, eingesetzt. Jeder dieser Algorithmen legt in Bezug auf die Entwicklung der Eintragsdichte, die Entwicklung der Eintragsverteilung und die Entwicklung der Eintragsgröße, der während der Durchführung einer HNF–Berechnung entstehenden Zwischenmatrizen, ein unterschiedliches Laufzeitverhalten an den Tag. Somit liegt die Idee nahe, wenn nicht ein einzelner der bekannten, publizierten Algorithmen in der Lage ist, unter vorgegebenen Rahmenbedingungen wie Laufzeit- und Hauptspeicherplatzrestriktionen die HNF einer großen, dünnbesetzten, ganzzahligen Matrix zu berechnen, es dennoch eine Kombination von HNF–Algorithmen, wobei ein Algorithmus einen anderen ablöst und auf dem Teilergebnis des vorangehenden aufsetzt, schaffen könnte.

Präziser formuliert bedeutet dies, wir bauen aus einzelnen HNF–Algorithmen einen *Hybrid–HNF–Algorithmus*, indem wir das Verhalten des gewünschten Algorithmus auf zwei getrennten Ebenen spezifizieren.

1. **Algorithmenebene:**
   Auf der Ebene der Algorithmen legen wir unter Berücksichtigung der gewünschten Rahmenbedingungen fest, welche HNF–Algorithmen zu welchem Zeitpunkt verwendet werden müssen, damit die Rahmenbedingungen nicht verletzt werden. Desweiteren legen wir den Übergangszeitpunkt fest. Dies beinhaltet die Bestimmung von entsprechenden Abbruch– bzw. Übergangskriterien.

2. **Implementierungsebene:**
   Auf der Ebene der Umsetzung bzw. Implementierung kommen weitere Festlegungen hinzu. Um effizient und mehr oder minder ressourcenschonend mit großen, dünnbesetzten Matrizen rechnen zu können, muß eine entsprechende Datenstruktur zur Ablage derselben implementiert werden. Die Auswahl der richtigen Datenstruktur ist

für die Effizienz der auf ihr operierenden Algorithmen von großer Bedeutung. Unterschiedliche HNF–Algorithmen führen unterschiedliche Operationen auf der zugrundeliegenden Datenstruktur aus. Somit sind für unterschiedliche HNF–Algorithmen unterschiedliche Datenstrukturen optimal. Deshalb muß zusätzlich zu der Entscheidung, wann welcher HNF–Algorithmus zum Einsatz kommt, festgelegt werden, wann welche zugrundeliegende Datenstruktur verwendet wird.

Dabei müssen erwartete Änderungen der Eintragsdichte, Eintragsverteilung und der Eintragsgröße berücksichtigt werden, um Einsparungseffekte bzgl. der Rechnenzeit nicht durch häufige Wechsel der Datenstruktur zunichte zu machen. Konkret bedeutet dies, daß man bei der Auswahl und Implementierung der verwendeten Datenstrukturen darauf achtet, einen nahtlosen Übergang von dünnbesetzten Ablagestrukturen in dichtbesetzte zu ermöglichen, ohne dabei dem Zwang zu unterliegen, alle Matrixelemente umspeichern zu müssen.

Unser Ziel besteht zusammenfassend darin, in Abhängigkeit von den Eigenschaften der speziellen, vorliegenden Problemstellung, d.h. den Eigenschaften der zu betrachtenden Matrizen, eine Folge von HNF–Algorithmen zu bestimmen und dabei detailliert festzulegen

- welcher Algorithmus,
- zu welchem Zeitpunkt,
- basierend auf welcher Datenstruktur

aufgerufen wird, um die HNF der Eingabematrix effizient unter Einhaltung der vorgegebenen Rahmenbedingungen zu berechnen. Die neuen HNF–Algorithmen, die wir im folgenden als *Hybrid–HNF–Algorithmen* bezeichnen, arbeiten dann gemäß der in Abbildung 4.1 schematisch dargestellten Vorgehensweise.

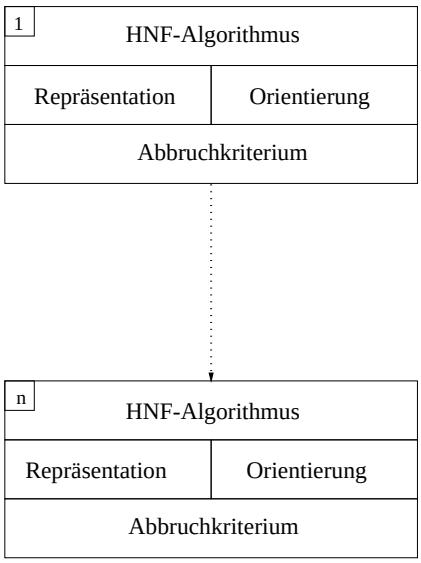


Abbildung 4.1: Schema der Hybrid–HNF–Algorithmen

Kapitel 5

Realisierung: Framework

Um die im vorangegangenen Abschnitt dargestellte Lösungsidee in die Praxis umzusetzen, wurde ein Framework bestehend aus drei Teilen entwickelt.

1. Erstens benötigen wir eine C++ Klassenstruktur, welche uns die nötigen Grundbausteine zur Konstruktion von Hybrid–HNF–Algorithmen bereitstellt. Konkret bedeutet dies, innerhalb der Klassenbibliothek LiDIA ein Framework von C++ Klassen zu designen und zu implementieren, das es erlaubt,

   - Operationen auf Matrizen unterschiedlicher Repräsentation (dichtbesetzt, dünnbesetzt, etc.),
   - mit hoher Ausführungseffizienz und
   - hoher Benutzerfreundlichkeit

   auszuführen. Damit schaffen wir die Grundlage, Algorithmen zur Berechnung der HNF angemessen implementieren und untersuchen zu können.

2. Zweitens müssen die bekannten publizierten Algorithmen inklusive eigener Varianten zur HNF–Berechnung in dem oben angesprochenen Framework implementiert und Daten über das Laufzeitverhalten der implementierten HNF–Algorithmen bezüglich der Kenngrößen

   (a) Entwicklung der Eintragsdichte,
   (b) Entwicklung der Eintragsgröße und
   (c) Laufzeit,

   gesammelt werden. Die gewonnenen Daten dienen uns im Rahmen der Anwendung unseres Frameworks als Hinweis dafür, in welcher Situation, welcher der implementierten Algorithmen auszuwählen ist. Sie entbindet nicht von der Aufgabe, für die jeweils speziell betrachtete Eingabematrixklasse eigene Optimierungen und eigene Versuchsreihen durchführen zu müssen.

3. Drittens benötigen wir eine Vorgehensweise, die festlegt, wie das Framework auf eine spezielle Problemstellung hin anzuwenden ist. Diese Fragestellung ist Gegenstand

des letzten Teils dieser Arbeit. In ihm beschreiben wir zunächst allgemein die Vorgehensweise zur Konstruktion von Hybrid–HNF–Algorithmen und illustrieren anhand von zwei Eingabematrixklassen, daß die Lösungsidee auch effizient in der Praxis funktioniert.

Die folgenden Teile dieser Arbeit beschreiben im Detail die Komponenten unseres Frameworks. Im folgenden dritten Teil beschreiben wir das Design und die Implementierung der Matrixklassen. Im vierten Teil stellen wir im Rahmen einer Systematik die Grundbausteine unseres Frameworks d.h. die bisher bekannten HNF–Algorithmen vor und illustrieren anschließend im fünften Teil eine mögliche Vorgehensweise zu Anwendung unseres Frameworks auf eine spezielle Problemstellung.

Teil III

Framework I: Design und Implementierung von Matrixklassen

Kapitel 6

Designprinzipien

In diesem Kapitel stellen wir die Designprinzipien und das Klassendesign unserer Implementierung von Matrizen in der Klassenbibliothek LiDIA vor.

Dazu benötigen wir die folgenden Begriffe [Str98]:

"Ein *abstrakter Datentyp* repräsentiert eine bestimmte Vorstellung eines Objekts und definiert sich aus Operationen, die dieser Datentyp bereitstellt. Er abstrahiert dabei von jedweder speziellen Implementierung. In dieser Hinsicht definiert ein abstrakter Datentyp ein Interface, das von jeder speziellen Realisierung implementiert werden muß."

"Eine *spezielle Implementierung* realisiert die Operationen des abstrakten Datentyps. Sie besteht aus einer Datenstruktur zur Speicherung der Datentypelemente und Operationen auf dieser Datenstruktur. In diesem Sinne bildet jede spezielle Implementierung des abstrakten Datentyps eine *Repräsentation* des Objekts."

Erklärtes Ziel des LiDIA–Projektes [BBP95] ist es, im Rahmen einer C++ Klassenbibliothek abstrakte Datentypen und Operationen anzubieten, die auf einem hohen Abstraktionslevel Lösungen für diverse Probleme der Zahlentheorie berechnen können.

Um dies zu erreichen, soll ein geeignetes C++ Klassendesign

- die Wiederverwendbarkeit des Sourcecodes,
- die leichte Erweiterbarkeit und Wartbarkeit der Gesamtbibliothek und
- die intuitive und damit leichte Benutzbarkeit der angebotenen Typen und deren Operationen

sicherstellen. Die Auswahl einer bestimmten Implementierungstechnik in Verbindung mit den Möglichkeiten der Programmiersprache C++ und die Auswahl der jeweiligen Algorithmen sollen hohe Effizienz mit geringer Sourcecodegröße und leichter Portabilität vereinen.

Diese Zielsetzungen mit besonderer Betonung auf der leichten Benutzbarkeit führen zu folgenden Designentscheidungen für die Realisierung von Matrizen in LiDIA :

1. **Hierarchische Template–Klassen**
2. **Dynamische Speicherverwaltung**
3. **Trennung von Datenstruktur und Interpretation**
4. **Trennung von Kernalgorithmen und Repräsentation**
5. **Sequenzorientierung**
6. **Trennung von Interface und Implementierung**

In den folgenden Abschnitten illustrieren wir anhand von einfachen Beispielen, was diese Designentscheidungen beinhalten und zeigen, welche Auswirkungen sie auf das Klassendesign der Matrizen in LiDIA haben.

6.1 Hierarchische Template–Klassen

Matrizen stellen ein Anordnungsschema für Elemente einer zugrundeliegenden Menge dar. Entstammen diese Elemente einer algebraischen Struktur, so induzieren die Verknüpfungen dieser Struktur Verknüpfungen, Operationen und Funktionen auf der Menge der zugehörigen Matrizen.

Diese Eigenschaft wurde in das Klassendesign der Matrixklassen durch die Verwendung von parametrisierten Klassen (Template–Klassen) in Kombination mit einer geeigneten Vererbungshierarchie integriert. Die Verwendung des Template–Mechanismus ermöglicht es, Operationen der Matrixklassen, die einmal im Sourcecode vorliegen, für verschiedene Argumenttypen zu instantiieren und somit dem Nutzer der Klassenbibliothek anzubieten. Die Integration einer Vererbungshierarchie ermöglicht die Staffelung von Funktionalitäten gemäß den Eigenschaften des Argumenttyps.

Aus diesem Blickwinkel betrachtet, haben wir es nicht nur mit einem abstrakten Datentyp zu tun, sondern, wie die Abbildung 6.1 zeigt, mit einer Vererbungshierarchie von abstrakten Datentypen, die in jeder Vererbungsstufe neuen Eigenschaften des zugrundeliegenden Elementtyps Rechnung tragen müssen.

6.2 Dynamische Speicherverwaltung

Für die aktuelle Implementierung der Matrizen in LiDIA haben wir uns für eine dynamische Speicherverwaltung entschieden. Dies hat den Vorteil, daß Matrizen unserer Klassenbibliothek in den Dimensionen schrumpfen und wachsen können. Der Nutzer muß nicht die Größe der verwendeten Matrizen fest in seinen Programmen verankern, sondern kann sie dynamisch mit entsprechenden Funktionen anpassen.

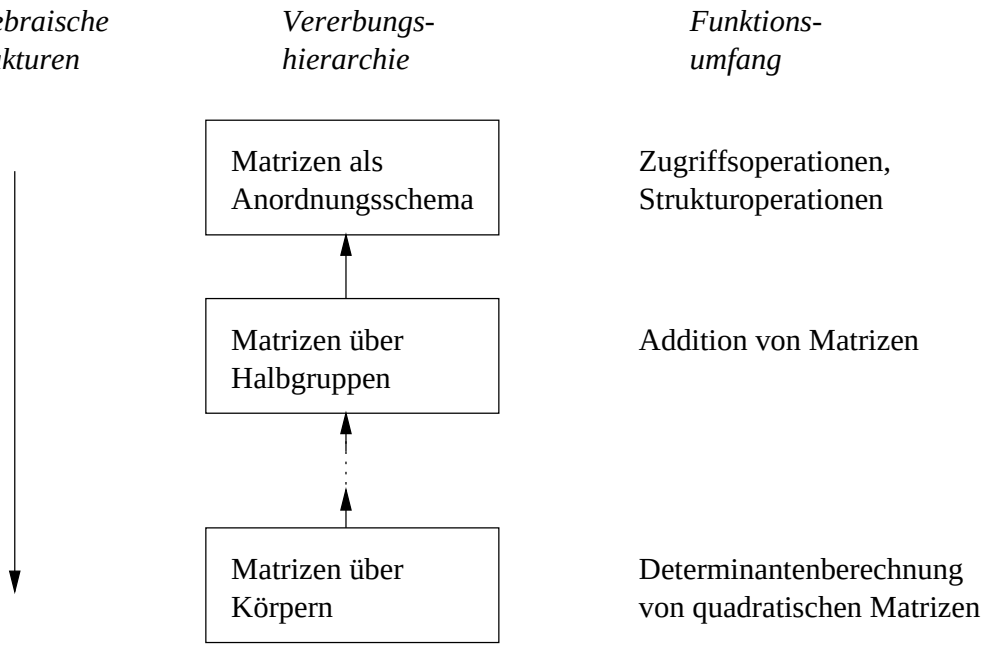


Abbildung 6.1: Hierarchische Template–Klassen

Der Hauptnachteil dieser Entscheidung liegt in der aufwendigen Programmierung der dynamischen Speicherverwaltung. Ein weiterer Nachteil ergibt sich daraus, daß wir die Dimensionen einer Matrix nicht zu effizienzsteigernden Optimierungen, wie Expression–Templates [Vel95a], Template–Metaprogramme [Vel95b], etc., nutzen können [Vel98, VP96].

6.1. Bemerkung
In [SL98c] wird gezeigt, wie man durch die Bildung von Blöcken die Effizienz von Expression–Templates mit der Flexibilität von Matrizen dynamischer Größe zumindest teilweise kombinieren kann. Eine solche Erweiterung ist für die hier vorgeschlagene Realisierung analog möglich, zur Zeit aber nicht umgesetzt.

6.3 Trennung von Datenstruktur und Interpretation

Algebraische Strukturen, wie z.B. Gruppen, Monoide, Ringe, etc., besitzen ein neutrales Element. Aufgrund der besonderen Eigenschaften dieses Elements ist es in einigen Anwendungen aus Gründen der Effizienzsteigerung und Speicherplatzminimierung sinnvoll, dieses ausgezeichnete Element aus der einer Repräsentation einer Matrix zugrundeliegenden Datenstruktur zu entfernen und lediglich Elemente, die ungleich diesem neutralen Element sind, abzuspeichern. Dies führt zur Unterscheidung von dicht- und dünnbesetzten Repräsentationen.

Will man zusätzlich die Kenntnis der Verteilung der verbleibenden Elemente zur Konstruktion effizienterer Algorithmen nutzen, erhalten wir eine weitere Unterteilungsebene. Man unterscheidet dann zum Beispiel zwischen:

- Diagonalmatrizen,
- Bandmatrizen,
- oberen Dreiecksmatrizen,
- unteren Dreiecksmatrizen, etc.

Nahezu alle Repräsentationen für Matrizen basieren auf der Ablage der Elemente in zweidimensionalen Arrays. Aus dieser Tatsache resultiert eine weitere Unterscheidungsebene. Gemäß der zugrundeliegenden Orientierung, d.h. der Reihenfolge in der Zeilenindex und Spaltenindex beim Zugriff auf ein Element der Datenstruktur verarbeitet werden, wird unterschieden in:

- Zeilenorientierung (Zeilenindex vor Spaltenindex)

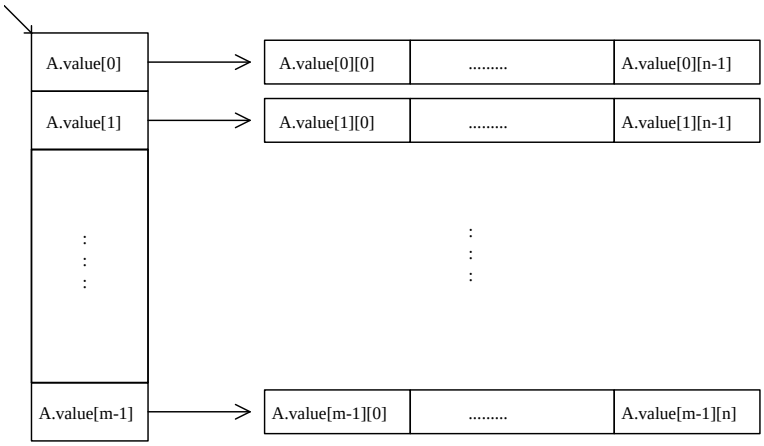


Abbildung 6.2: Zeilenorientierte Elementspeicherung

- Spaltenorientierung (Spaltenindex vor Zeilenindex)

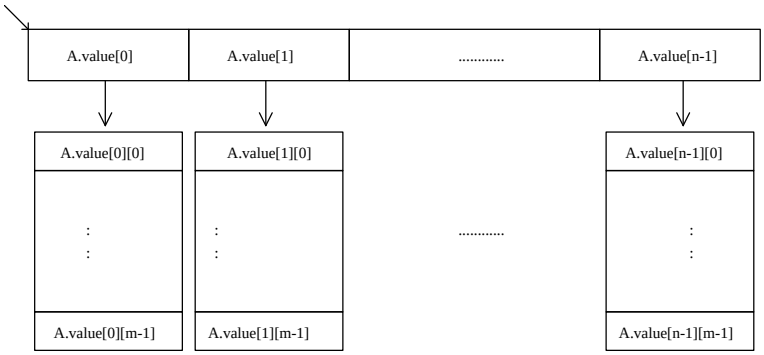


Abbildung 6.3: Spaltenorientierte Elementspeicherung

Faßt man die oben aufgeführten Unterscheidungsebenen zusammen, so führt dies kombinatorisch zu einer großen Anzahl von möglichen Repräsentationen für Matrizen.

In der aktuellen LiDIA –Implementierung werden vier Repräsentationen unterstützt. Dies sind:

1. die “klassische” Repräsentation, d.h. dichtbesetzte, zeilenorientierte Repräsentation,
2. die dichtbesetzte, spaltenorientierte Repräsentation,
3. die dünnbesetzte, zeilenorientierte Repräsentation und schließlich
4. die dünnbesetzte, spaltenorientierte Repräsentation.

Die Integration einer dünnbesetzten Repräsentation in die Realisierung der Matrixklassen ist aus dem Blickwinkel der Effizienz und der Speicherplatzoptimierung eine Grundvoraussetzung für den Umgang mit großen, dünnbesetzten Matrizen und somit essentiell für unsere Zielsetzung.

In der aktuellen Implementierung der Matrizen in LiDIA haben wir uns entschlossen, lediglich *eine* Ablageform zur Realisierung der dünnbesetzten Repräsentationen zu verwenden und nicht mehrere, wie dies zum Beispiel in [PRL95] vorgeschlagen wird. Auf Details unserer Ablagestrategie gehen wir zu einem späteren Zeitpunkt näher ein. Deshalb verzichten wir an dieser Stelle auf weitere Einzelheiten. Betonen möchten wir an dieser Stelle noch, daß es durchaus möglich ist, weitere Ablageformen und somit weitere Repräsentationen zu integrieren.

Richten wir unser Augenmerk nun auf verschiedene Möglichkeiten, Matrizen mit mehreren Repräsentationen in eine einzige Klassenstruktur zu integrieren. Dabei spielt der folgende Aspekt in unseren Überlegungen eine wesentliche Rolle:

Wie wir bereits in den vorangegangenen Abschnitten gesehen haben, zeichnen sich die durchgeführten HNF–Berechnungen durch eine Zunahme der Eintragsdichte aus. Ausgehend von einer dünnbesetzten Matrix erhöht sich die Eintragsdichte der noch verbleibenden Matrix hin zu einer dichtbesetzten Matrix. Aus diesem Grund sollte die verwendete Implementierung als zusätzliche Nebenbedingung einen quasi nahtlosen Übergang ermöglichen, d.h. der Übergang von einer Repräsentation in eine andere sollte ohne Kopieren aller Einträge vonstatten gehen.

Zur Implementierung der Repräsentationsvielfalt bieten sich folgende Möglichkeiten an:

1. Für jede Spezialrepräsentation definieren wir eine eigene Datenstruktur und kapseln bzw. verwalten sie durch eine entsprechende Repräsentationsklasse. Der Nachteil dieser Technik liegt in der Vielzahl der Cast–Funktionen bzw. Konstruktoren, die benötigt werden, um eine Repräsentation in eine andere umzuwandeln. Ein nahtloser Übergang von einer dünnbesetzten in eine dichtbesetzte Repräsentation ist damit nicht möglich.
2. Für die Implementierung in LiDIA verwenden wir einen anderen Ansatz. In einer Basisklasse definieren wir eine allgemeine, repräsentationsunabhängige Datenstruktur. Abgeleitete Repräsentationsklassen interpretieren nun diese Datenstruktur in der jeweiligen repräsentationsspezifischen Art und Weise, indem sie die Zugriffsoperationen gemäß der gewünschten Repräsentation implementieren.

   Aufgrund dessen, daß alle Operationen auf der gleichen Datenstruktur ausgeführt werden, ist ein nahtloser Übergang von einer dünnbesetzten Repräsentation über eine Mischrepräsentation hin zu einer dichtbesetzten Repräsentation innerhalb einer

Orientierung ohne den Aufruf von Cast-Operatoren bzw. Konstruktoren möglich. Dadurch erwarten wir einen Effizienzgewinn bei diversen Matrixalgorithmen.

6.4 Trennung von Kernalgorithmen und Repräsentation

Die Designentscheidung, mehrere Repräsentationen zu unterstützen, und der Wunsch, die Wartung der Matrixklassen so einfach wie möglich zu halten und ein Optimum an Effizienz zu erreichen, scheinen auf den ersten Blick unvereinbar zu sein.

Zur Erreichung einer optimalen Ausführungseffizienz auf der Basis einer aktuellen Matrixrepräsentation ist es erforderlich, den gewünschten Algorithmus speziell an die Repräsentation anzupassen. Auf den ersten Blick betrachtet, führt diese Beobachtung dazu, daß der jeweilige Algorithmus mehrfach implementiert wird und dadurch der Umfang des Quellcodes erheblich ansteigen würde.

Eine speziell vor diesem Hintergrund entwickelte Programmiertechnik, auf die wir im nächsten Kapitel eingehen werden, löst dieses Problem. Die Kernidee besteht darin, repräsentationsabhängige Algorithmenteile von anderen Teilen zu trennen und jeweils in Template–Klassen zu kapseln. Der Compiler führt zur Übersetzungszeit die repräsentationsabhängigen mit den repräsentationsunabhängigen Teilen zusammen und erzeugt eine speziell an die Repräsentation angepaßte Implementierung.

Die repräsentationsunabhängigen Teile eines Algorithmus, ergänzt um Template–Funktionsaufrufe für die repräsentationsabhängigen Teile, bilden den **Kernalgorithmus**. Die verbleibenden repräsentationsabhängigen Teile eines Algorithmus bilden die **Modulalgorithmen**.

6.2. Bemerkung
Im allgemeinen setzt sich eine spezielle Implementierung aus einem Kernalgorithmus und mehreren Modulalgorithmen zusammen. Modulalgorithmen, die zu einer Repräsentation gehören, fassen wir in **Modulklassen** zusammen.

6.5 Sequenzorientierung

Im Vorfeld und während der Entstehung dieser Arbeit wurden eine Vielzahl anderer Implementierungen von Matrixklassen untersucht und analysiert. Die Mehrzahl der betrachteten Implementierungen versuchen lediglich die einzelne Operation an sich effizient zu realisieren. Dabei vernachlässigen sie aber Informationen, die dazu dienen könnten, die Ausführungsgeschwindigkeit einer Sequenz von Matrixoperationen zu optimieren.

Diese Einschränkung wird in der aktuellen Implementierung der Matrixklassen in LiDIA teilweise eliminiert. Zur Zeit werden Informationen über den Rang, über die linear unabhängigen Zeilen bzw. Spalten und über die Verteilung der Nicht–Nulleinträge einer Matrix gespeichert und bei Bedarf zur Laufzeitoptimierung herangezogen.

6.3. Bemerkung
Die Erweiterung der Matrixdatenstruktur erhöht den Speicherplatzbedarf einer Matrixvariablen nur unwesentlich, da nur für "einfache" Informationen Platz vorgesehen ist.

6.6 Trennung von Interface und Implementierung

Die Verwendung von Interface–Klassen ist ein Hauptbestandteil der komponentenorientierten Programmierung. Sie stellen eine Art *Vertrag* zwischen dem Anbieter und dem Nutzer einer Komponenten dar. Halten sich beide an die im Interface festgelegten Aufrufkonventionen, ist die Austauschbarkeit der Komponente sichergestellt und die Benutzung verschiedener Komponenten mit gleichem Interface sehr einfach.

In LiDIA verfolgen wir durch die Integration von Interface–Klassen ähnliche Ziele. Erstens wollen wir den Nutzer nicht mit Details der jeweiligen Matrixrepräsentation belasten. Zweitens soll die Nutzbarkeit der abstrakten Klassen der Vererbungshierarchie erhöht werden. Drittens soll der Anwender der Klassenbibliothek LiDIA von der Aufgabe entbunden werden, für jeden Typ, den er als Template–Argument innerhalb der Matrixklassen benutzen möchte, die entsprechende algebraische Struktur und somit die adäquate Template–Matrixklasse kennen zu müssen (vgl. [PRL95, Pap97]).

Wir unterscheiden die folgenden zwei Interface–Typen:

1. **dynamisches Interface:** Die parametrisierte C++ Klasse `matrix` bildet innerhalb des Klassendesigns der Matrixklassen ein dynamisches Interface. Dies bedeutet, daß die Repräsentation der jeweiligen Matrix in Abhängigkeit von den benutzten Algorithmen zur Laufzeit wechseln kann.

2. **statisches Interface:** Im Gegensatz dazu bilden die parametrisierten C++ Klassen `dense_matrix` bzw. `sparse_matrix` innerhalb der Matrixklassen ein dichtbesetztes bzw. ein dünnbesetzes statisches Interface. Die Repräsentation der Matrix ist festgelegt und wird während der Laufzeit nicht verändert.

6.4. Bemerkung
Die Interface–Klassen sind als reine Inline–Template–Klassen konzipiert und erzeugen somit keinen Laufzeitoverhead.

Kapitel 7

Programmiermethodik

Die im vorangegangenen Kapitel erläuterten Designentscheidungen implizieren spezielle Anforderungen an die zu verwendenden Programmiertechniken. Insbesondere die *Trennung von Datenstruktur und Repräsentation* und die *Trennung von Kernalgorithmus und Repräsentation* führen zu Problemstellungen, die einer genaueren Untersuchung bedürfen. Dabei kristallisieren sich folgende Fragestellungen heraus:

Welche C++ Implementierungstechnik ist am besten geeignet, um mit optimaler Ausführungseffizienz

- einen festen Kernalgorithmus mit variablen Algorithmenmodulen zu versehen, die den Kernalgorithmus an die jeweilige Matrixrepräsentation anpassen?
- einen festen Kernalgorithmus mit variablen Algorithmenmodulen zu versehen, die eine Vielzahl von Varianten des Kernalgorithmus mit einem Minimum an zusätzlichem Code realisieren?

In der Programmiersprache C++ existieren mehrere Möglichkeiten, die gewünschte Funktionalität zu realisieren. Im folgenden Abschnitt gehen wir kurz auf die jeweilige Technik ein und erläutern die Vor- und Nachteile in Bezug auf unsere Zielsetzung.

Zur Veranschaulichung illustrieren wir die jeweilige Realisierungstechnik anhand des folgenden einfachen Beispiels.

7.1. Beispiel

Zur Realisierung von Matrizen variabler Dimension über dem Datentyp `double` verwenden wir die folgende C++ Klasse:

```
class matrix
{
private:

  int no_of_rows;     // Anzahl der Zeilen
```

```
  int no_of_columns;  // Anzahl der Spalten
  double **value;     // Wertearray
  bool SW;

public:

  matrix();
  ~matrix();

  friend void add(matrix &, const matrix &, const matrix &);
};
```

Es ist leicht zu sehen, daß die obige Datenstruktur eine zeilenweise sowie eine spaltenweise Ablage der Einträge ermöglicht. Nehmen wir also an, daß die Konstruktoren und Destruktoren für diese Klasse bereits implementiert seien und die boolesche Member–Variable `SW` zur Kodierung der Ablagestrategie benutzt wird. (`SW = true` bedeutet zeilenweise und `SW = false` spaltenweise Ablage)

Unser Ziel ist es, diese Klasse um die Funktion

```
void add(matrix &A, const matrix &B, const matrix &C),
```

welche die Matrixaddition $A = B + C$ realisiert, zu erweitern, wobei wir sowohl eine zeilenweise als auch eine spaltenweise Ablage der Matrizen unterstützen wollen.

1. **Codereplikation**
   Die erste Möglichkeit, die gewünschte Funktionalität zu erreichen, ist die Codereplikation. In dieser Variante werden die Funktionen in allen Details voll ausprogrammiert.

   7.2. Beispiel (Fortsetzung des Beispiels 7.1)
   Angewendet auf unsere Beispielklasse bedeutet dies, daß für alle Parameterkombinationen von zeilenweiser und spaltenweiser Ablage jeweils getrennt eine Funktion in der C++ Klasse `matrix` implementiert wird. Mittels einer switch–Anweisung wird dann in der Funktion `add` entschieden, welche dieser Funktionen aufgerufen wird. Wir erhalten somit folgende Sourcecodeteile:

   ```
   void matrix::ZZZ_add(const matrix &B, const matrix &C)
   {
     //
     // Matrix A: zeilenweise Ablage
     // Matrix B: zeilenweise Ablage
     // Matrix C: zeilenweise Ablage
     //

     for (int i = 0; i < A.no_of_rows; i++)
       for (int j = 0; j < A.no_of_columns; j++)
   ```

```
      value[i][j] = B.value[i][j] + C.value[i][j];
}

void matrix::SZZ_add(const matrix &B, const matrix &C)
{
  //
  // Matrix A: spaltenweise Ablage
  // Matrix B: zeilenweise Ablage
  // Matrix C: zeilenweise Ablage
  //

  for (int i = 0; i < A.no_of_rows; i++)
    for (int j = 0; j < A.no_of_columns; j++)
      value[j][i] = B.value[i][j] + C.value[i][j];
}

usw.

void matrix::SSS_add(const matrix &B, const matrix &C)
{
  //
  // Matrix A: spaltenweise Ablage
  // Matrix B: spaltenweise Ablage
  // Matrix C: spaltenweise Ablage
  //

  for (int i = 0; i < A.no_of_rows; i++)
    for (int j = 0; j < A.no_of_columns; j++)
      value[j][i] = B.value[j][i] + C.value[j][i];
}

friend void add(matrix &A, const matrix &B, const matrix &C)
{
  if (A.SW == true)
    {
      if (B.SW == true)
        if (C.SW == true)
          A.ZZZ_add(B, C);
        else
          A.ZZS_add(B, C);
      else
        if (C.SW == true)
          A.ZSZ_add(B, C);
        else
          A.ZSS_add(B, C);
    }
  else
    {
      if (B.SW == true)
```

```
        if (C.SW == true)
          A.SZZ_add(B, C);
        else
          A.SZS_add(B, C);
      else
        if (C.SW == true)
          A.SSZ_add(B, C);
        else
          A.SSS_add(B, C);
    }
}
```

Diese Methode liefert eine sehr hohe Ausführungseffizienz der einzelnen Funktionen, führt aber zu einer großen Codebasis und somit zu hohen Kosten, diesen Code zu warten.

Zu Verdeutlichung ist in der Abbildung 7.1 die Programmiertechnik der Codereplikation nochmals schematisch dargestellt.

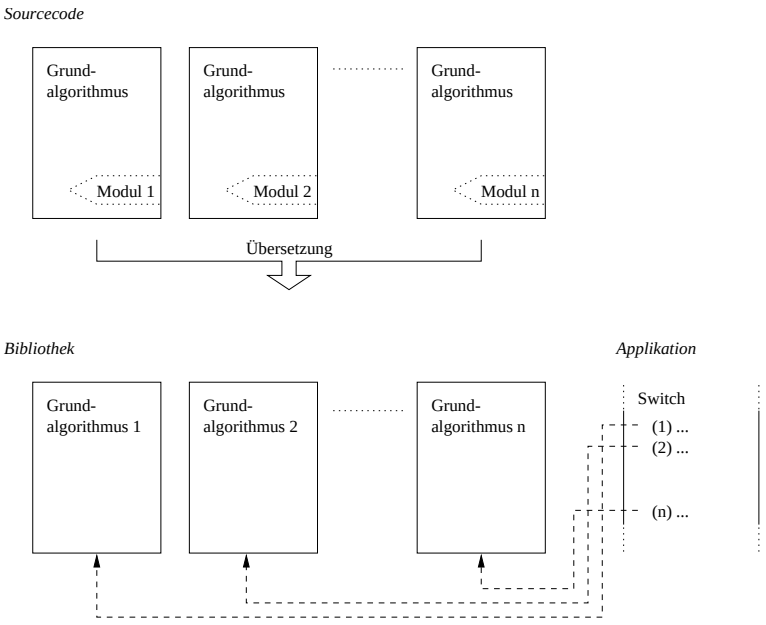


Abbildung 7.1: Schema: Codereplikation

Diese Technik ist dann sinnvoll, wenn die jeweiligen Funktionen überwiegend aus repräsentationsabhängigem Code bestehen. In diesem Fall überwiegt der Vorteil der leichten Lesbarkeit gegenüber der Codereduktion.

2. **Modulare Programmierung**
   Eine zweiten Möglichkeit, die gewünschte Funktionalität zu erreichen, bezeichnen wir als *modulare Programmierung*. In dieser Methode wird das Kernprogramm von repräsentationsabhängigen Teilen getrennt. Über eine entsprechende Abfrage von Flags wird direkt an der jeweiligen Stelle im Kernalgorithmus überprüft, welche der Modulalgorithmen benötigt werden.

7.3. Beispiel (Fortsetzung des Beispiels 7.1)
Für unsere Beispielklasse bedeutet dies, daß wir zunächst spezielle Zugriffsfunktionen zum Lesen und Schreiben in zeilenweiser bzw. spaltenweiser Ablage implementieren. Anschließend realisieren wir die Funktion `add`, wobei wir bei jedem lesenden und schreibenden Zugriff entscheiden, welche der Zugriffsfunktionen wir verwenden, d.h. welche Orientierung die jeweilige Matrixablage besitzt. Wir erhalten somit folgende Sourcecodeteile:

```
//
// Lesen
//

double matrix::zeilenorientiert_member(int i, int j) const
{return value[i][j];}

double matrix::spaltenorientiert_member(int i, int j) const
{return value[j][i];}

//
// Schreiben
//

void matrix::zeilenorientiert_sto(int i, int j, double w) const
{value[i][j] = w;}

void matrix::spaltenorientiert_sto(int i, int j, double w) const
{value[j][i] = w;}

friend void add(matrix &A, const matrix &B, const matrix &C)
{
  double b, c;
  for (int i = 0; i < A.no_of_rows; i++)
    for (int j = 0; j < A.no_of_columns; j++)
      {
        // Lesen Matrix B
        if (B.SW == true)
          b = B.zeilenorientiert_member(i,j);
        else
          b = B.spaltenorientiert_member(i,j);

        // Lesen Matrix C
        if (C.SW == true)
          c = C.zeilenorientiert_member(i,j);
        else
          c = C.spaltenorientiert_member(i,j);

        // Schreiben Matrix A
        if (A.SW == true)
          A.zeilenorientiert_sto(i,j,b+c);
```

```
        else
          A.spaltenorientiert_sto(i,j,b+c);
      }
}
```

Diese Methode führt im Vergleich zur ersten Technik zu einer wesentlich geringeren Codebasis und somit zu einer leichteren Wartbarkeit. Diesen Vorteil bezahlen wir durch viele zusätzliche Überprüfungen und Funktionsaufrufe. Ein Inlining der Modulfunktionen ist möglich und kann zur Eliminierung von Funktionsaufrufen und somit zur Effizienzsteigerung eingesetzt werden.

In der Abbildung 7.2 ist die hier vorgestellte Programmiertechnik nochmals schematisch dargestellt.

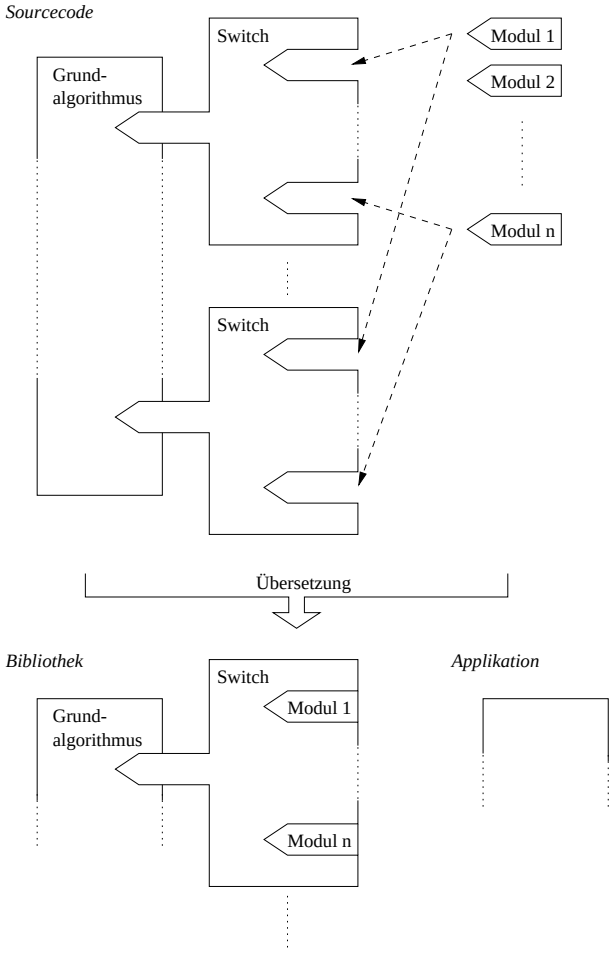


Abbildung 7.2: Schema: Modulare Programmierung

3. **Funktionspointer**
   Um nun die Anzahl der zusätzlichen Überprüfungen zu minimieren, bietet sich die Verwendung von Funktionspointern an. Bei dieser Methode werden zu Beginn des Kernalgorithmus Funktionspointer auf die jeweils gewünschten Modulfunktionen gesetzt. Somit kann im weiteren Verlauf des Kernalgorithmus über diese Funktionspointer der richtige Modulalgorithmus aufgerufen werden.

7.4. Beispiel (Fortsetzung des Beispiels 7.1)
Analog zur Programmiertechnik der *modularen Programmierung* implementieren wir wiederum zunächst spezielle Zugriffsfunktionen. In der Funktion `add` setzen wir zu Beginn entsprechend der jeweiligen Orientierung der Matrixablage Funktionspointer auf die jeweilige Zugriffsfunktion und implementieren anschließend den eigentlichen Additionsalgorithmus auf der Basis dieser Funktionspointer. Dies führt zu folgenden Sourcecodeteilen:

```
//
// Lesen
//

double matrix::zeilenorientiert_member(int i, int j) const
{return value[i][j];}

double matrix::spaltenorientiert_member(int i, int j) const
{return value[j][i];}

//
// Schreiben
//

void matrix::zeilenorientiert_sto(int i, int j, double w) const
{value[i][j] = w;}

void matrix::spaltenorientiert_sto(int i, int j, double w) const
{value[j][i] = w;}

friend void add(matrix &A, const matrix &B, const matrix &C)
{
  double b, c;
  double (* Bmember)(int, int);
  double (* Cmember)(int, int);
  void (* Asto)(int, int, double);

  // Lesen Matrix B
  Bmember = (B.SW == true ? &(B.zeilenorientiert_member) :
                            &(B.spaltenorientiert_member);

  // Lesen Matrix C
  Cmember = (C.SW == true ? &(C.zeilenorientiert_member) :
                            &(C.spaltenorientiert_member);

  // Schreiben Matrix A
  Asto = (A.SW == true ? &(A.zeilenorientiert_sto) :
                         &(A.spaltenorientiert_sto);

  // Additionsalgorithmus
  for (int i = 0; i < A.no_of_rows; i++)
```

```
    for (int j = 0; j < A.no_of_columns; j++)
       Asto(i,j,Bmember(i,j)+Cmember(i,j));
}
```

Die Vorteile liegen im Vergleich zur ersten Implementierungstechnik in einer kleineren Codebasis und im Vergleich zur zweiten Implementierungstechnik in einer verminderten Anzahl an zusätzlichen Flagüberprüfungen. Der Nachteil dieser Technik liegt darin, daß die Verwendung von Funktionspointern ein *Inlining* der Modulfunktionen an den jeweiligen Aufrufstellen im Kernalgorithmus verhindert, was sich in einer verminderten Effizienz im Vergleich zur ersten Implementierungstechnik äußert.

In der Abbildung 7.3 haben wir wiederum die beschriebene Implementierungstechnik schematisch dargestellt.

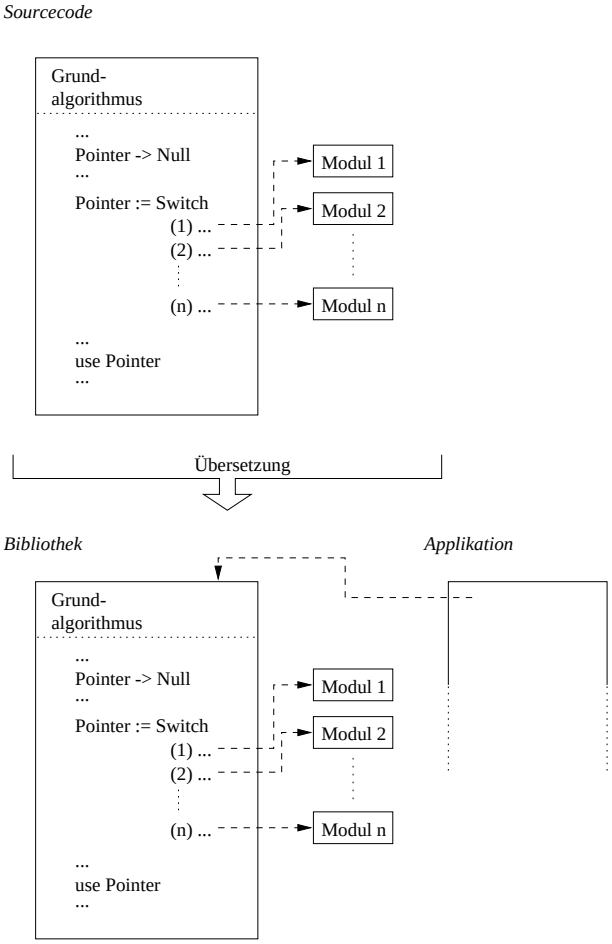


Abbildung 7.3: Schema: Funktionspointer

4. **Basisklasse mit virtuellen Funktionen**
   C++ als objektorientierte Programmiersprache [Str92, Str98] bietet für die von uns angestrebte Funktionalität eine weitere Lösungsmöglichkeit an: *Basisklasse mit virtuellen Funktionen*

Bei dieser Programmiertechnik werden die Modulfunktionen als virtuelle Funktionen in Basisklassen deklariert und somit im Bezug auf Anzahl und Typ der Aufrufparameter festgelegt. Von diesen Basisklassen abgeleitete Klassen implementieren diese Funktionen in einer jeweils der gewünschten Repräsentation bzw. Teilfunktionalität angebrachten Art und Weise. Der Kernalgorithmus verwendet Funktionen aus der virtuellen Basisklasse und ruft diese über einen Pointer auf die Basisklasse auf. Um verschiedene Repräsentationen bzw. verschiedene Algorithmenvarianten zu unterstützen, wird der Pointer auf die Basisklasse auf die jeweils gewünschte abgeleitete Klasse umgesetzt, die die jeweiligen Modulimplementierungen der gewünschten Repräsentation bzw. Funktionalität beinhaltet.

7.5. Beispiel (Fortsetzung des Beispiels 7.1)
Die Verwendung der C++ Klasse `matrix` als virtuelle Basisklasse führt dazu, daß die Ablagestrategie explizit, d.h. für den Nutzer der Matrixklasse im Klassennamen sichtbar gemacht wird. Von der Klasse `matrix` werden die C++ Klassen `row_oriented_matrix` und `column_oriented_matrix` abgeleitet, indem dem Klassennamen entsprechend lesende und schreibende Zugriffsfunktionen implementiert werden.

```
class matrix
{
private:

  int no_of_rows;     // Anzahl der Zeilen
  int no_of_columns;  // Anzahl der Spalten
  double **value;     // Wertearray
  bool SW;

public:

  matrix();
  ~matrix();

  virtual double member(int, int) = 0;
  virtual void sto(int, int, double) = 0;

  friend void add{matrix &, const matrix &, const matrix &);
};

class row_oriented_matrix : public class matrix
{
public:

  row_oriented_matrix();
  ~row_oriented_matrix();

  double member(int i, int j)
    {
      return value[i][j];
```

```
    }

  void sto(int i, int j, double w)
    {
      value[i][j] = w;
    }
};

class column_oriented_matrix : public class matrix
{
public:

  column_oriented_matrix();
  ~column_oriented_matrix();

  double member(int i, int j)
    {
      return value[j][i];
    }

  void sto(int i, int j, double w)
    {
      value[j][i] = w;
    }
};
```

Die Additionsfunktion selbst wird in der Klasse `matrix` wie folgt abgelegt, wobei der Mechanismus der virtuellen Vererbung dafür sorgt, daß die richtige, der Ablagestrategie entsprechende Member–Funktion aufgerufen wird.

```
friend void add(matrix &A, const matrix &B, const matrix &C)
{
  for (int i = 0; i < A.no_of_rows; i++)
    for (int j = 0; j < A.no_of_columns; j++)
       A.sto(i,j,B.member(i,j)+C.member(i,j));
}
```

Diese Technik ähnelt der im vorangegangenen Punkt angesprochenen. Der Vorteil dieser Methode liegt in der kleinen Codebasis und somit in der leichten Wartbarkeit des Sourcecodes. Desweiteren bietet diese Technik gegenüber der vorherigen den Vorteil, daß die Integration weiterer Repräsentationen bzw. Algorithmenvarianten leicht möglich ist. Der Nachteil dieser Technik liegt darin, daß ein Inlining der Funktionen aufgrund der Laufzeitabhängigkeit nicht möglich ist und somit in einen Effizienzverlust mündet.

In der Abbildung 7.4 ist die Idee dieser Implementierungstechnik nochmals angedeutet.

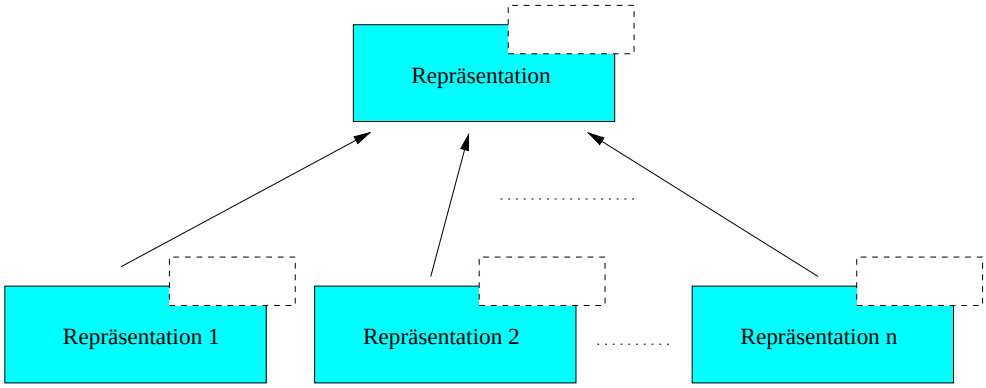


Abbildung 7.4: Schema: Basisklasse mit virtuellen Funktionen

5. **Template Kernel / Template Modul**
   Vor dem Hintergrund, daß keine der bisher vorgestellten Implementierungstechniken unseren Anforderungen, insbesondere für kleine repräsentations- bzw. funktionalitätsabhängige Modulteile, gerecht wird, wurde in Zusammenarbeit mit Werner Backes und Susanne Wetzel eine neue Implementierungstechnik entwickelt [WTB97, TWB97, Bac98].

   Sie basiert auf dem Zusammenbau und der Übersetzung von repräsentations- bzw. funktionalitätsabhängigen Algorithmen zur Compilezeit aus einem repräsentations- bzw. funktionalitätsunabhängigen Kernalgorithmus und repräsentations- bzw. funktionalitätsabhängigen Modulalgorithmen. Dabei machen wir uns die C++ Technik der Template–Klassen und die Fähigkeit des Inlining moderner Compiler zunutze.

7.6. Beispiel (Fortsetzung des Beispiels 7.1)
Angewendet auf unser Beispiel bedeutet dies, daß wir in einer Template–Kernel–Klasse den eigentlichen Kernalgorithmus, d.h. den Additionsalgorithmus, ablegen.

```
template < class Modul_A, class Modul_B, class Modul_C >
class add_kernel
{
private:

   Modul_A mod_A;
   Modul_B mod_B;
   Modul_C mod_C;

public:

   add_kernel(){}
   ~add_kernel(){}

  void add(matrix &A, const matrix &B, const matrix &C);
};
```

Dabei ist der Kernalgorithmus wie folgt implementiert:

```
template < class Modul_A, class Modul_B, class Modul_C >
void add_kernel < Modul_A, Modul_B, Modul_C > ::
add(matrix &A, const matrix &B, const matrix &C)
{
  for (int i = 0; i < A.no_of_rows; i++)
    for (int j = 0; j < A.no_of_columns; j++)
       mod_A.sto(A,i,j,mod_B.member(B,i,j) + mod_C.member(C,i,j));
}
```

In den Modulklassen `row_oriented_matrix` und `column_oriented_matrix` werden die dem Klassennamen entsprechenden lesenden und schreibenden Zugriffsfunktionen implementiert.

```
class row_oriented_matrix
{
public:

  row_oriented_matrix(){}
  ~row_oriented_matrix(){}

  double member(const matrix &A, int i, int j)
    {return A.value[i][j];}

  void sto(matrix &A, int i, int j, double w)
    {A.value[i][j] = w;}
};

class column_oriented_matrix
{
public:

  column_oriented_matrix(){}
  ~column_oriented_matrix(){}

  double member(const matrix &A, int i, int j)
    {return A.value[j][i];}

  void sto(matrix &A, int i, int j, double w)
    {A.value[j][i] = w;}
};
```

Dabei manipulieren sie die Datenstruktur unsere Matrixklasse. Dies erfordert die Anbindung dieser Template–Modul–Klassen als “`friend`”.

```
#define ROM row_oriented_matrix
#define COM column_oriented_matrix

class matrix
{
```

```
private:

  int no_of_rows;     // Anzahl der Zeilen
  int no_of_columns;  // Anzahl der Spalten
  double **value;     // Wertearray
  bool SW;

public:

  matrix();
  ~matrix();

  friend void add{matrix &, const matrix &, const matrix &);

  // friend Klassen
  friend class ROM;
  friend class COM;
  friend class add_kernel < ROM,ROM,ROM >;

  friend class add_kernel < COM,ROM,ROM >;
  friend class add_kernel < ROM,COM,ROM >;
  friend class add_kernel < ROM,ROM,COM >;

  friend class add_kernel < COM,COM,ROM >;
  friend class add_kernel < COM,ROM,COM >;
  friend class add_kernel < ROM,COM,COM >;

  friend class add_kernel < COM,COM,COM >;
};
```

Zur Vervollständigung unseres Beispiels geben wir nun noch an, wie die jeweiligen Algorithmen, die aus der Kombination des Kernalgorithmus mit den verschiedenen Modulbausteinen entstehen, zusammengebaut und innerhalb der Funktion `add` aufgerufen werden.

```
#define ROM row_oriented_matrix
#define COM column_oriented_matrix

friend void add(matrix &A, const matrix &B, const matrix &C)
{
  if (A.SW == true)
    {
      if (B.SW == true)
        if (C.SW == true)
          {
            add_kernel < ROM,ROM,ROM > modul;
            modul.add(A, B, C);
          }
        else
```

```
            {
              add_kernel < ROM,ROM,COM > modul;
              modul.add(A, B, C);
            }
        else
          if (C.SW == true)
            {
              add_kernel < ROM,COM,ROM > modul;
              modul.add(A, B, C);
            }
          else
            {
              add_kernel < ROM,COM,COM > modul;
              modul.add(A, B, C);
            }
      }
    else
      {
        if (B.SW == true)
          if (C.SW == true)
            {
              add_kernel < COM,ROM,ROM > modul;
              modul.add(A, B, C);
            }
          else
            {
              add_kernel < COM,ROM,COM > modul;
              modul.add(A, B, C);
            }
        else
          if (C.SW == true)
            {
              add_kernel < COM,COM,ROM > modul;
              modul.add(A, B, C);
            }
          else
            {
              add_kernel < COM,COM,COM > modul;
              modul.add(A, B, C);
            }
      }
}
```

Die Vorteile dieser neuen Technik liegen in der kleinen und damit leicht zu wartenden Codebasis. Ist der Compiler in der Lage, Inlining zu verwenden, führt diese Technik zu einer optimalen Ausführungseffizienz im Verhältnis zur Codereplikation. Ist der Compiler dazu nicht oder nur teilweise dazu in der Lage, liegt die Ausführungseffizienz im Bereich der modularen Programmierung. Der Hauptnachteil dieser Technik liegt in den langen Compilezeiten. Desweiteren ist das Design von Algorithmen auf

der Basis dieser Technik gewöhnungsbedürftig, entspricht nicht dem Standard und wird somit nicht originär von Tools unterstützt.

In der Abbildung 7.5 ist diese Programmiertechnik nochmals schematisch dargestellt.

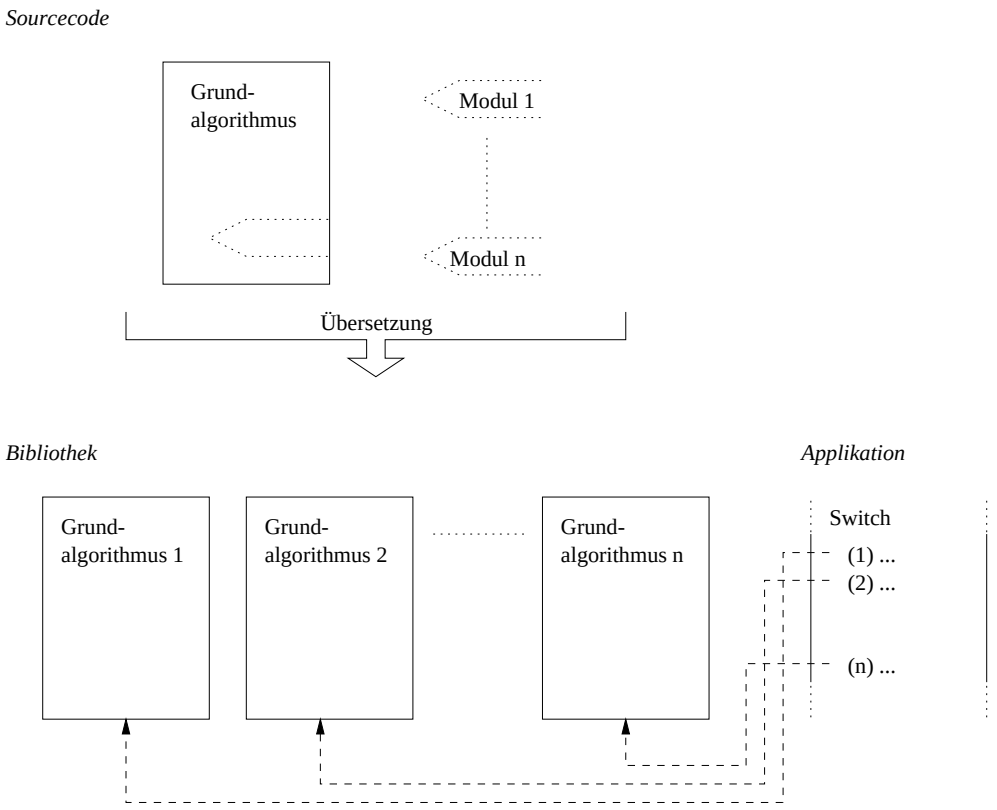


Abbildung 7.5: Schema: Template Modul / Template Kernel

Die Vorgehensweise, Algorithmen zur Compilezeit zusammenzubauen, rückt diese Implementierungstechnik in die Nähe der *Generischen Programmierung* und damit in Richtung der Standard Template Library (STL). Die Kernidee, auf der die STL basiert, besteht darin, daß viele Algorithmen von der speziellen Datenstruktur, auf der sie operieren, abstrahiert werden können. Die Algorithmen benötigen typischerweise lediglich die Fähigkeit, die Datenstruktur zu traversieren und auf ein Element der Datenstruktur zugreifen zu können. Zu diesem Zweck werden in der STL Iteratoren eingesetzt, die diese Schnittstelle realisieren (vgl. [Str98]).

Unser Ansatz ist wesentlich allgemeiner und beinhaltet die Vorgehensweise der STL als Teilaspekt, oder anders ausgedrückt, wir iterieren die Vorgehensweise der STL über mehrere Stufen hinweg. In LiDIA werden zur Compilezeit Algorithmen in unterschiedlichen Varianten basierend auf mehreren Repräsentationen zusammengesetzt und zur Verfügung gestellt.

7.7. Bemerkung

Neuere Entwicklungen wie die Matrix Template Library [SLL98, SL98a, SL98b, SL98c, LSL99] basieren auf einer Realisierungsidee, die Ähnlichkeiten mit der hier

vorgestellten aufweist. Dies belegt, daß mit der stetigen Weiterentwicklung der Compiler, insbesondere im Hinblick auf die Handhabung von Template–Klassen und Inline–Vorschlägen, der intensiven Verwendung von Template–Konstruktionen zur Sourcecodeminimierung und Funktionalitätsmaximierung nichts mehr im Wege steht.

Kapitel 8

Matrixklassen in LiDIA

In den vorangegangenen Kapiteln haben wir die Designentscheidungen und Implementierungstechniken vorgestellt, die das Klassendesign der Matrizen in LiDIA und deren Implementierung beeinflußt haben. In diesem Kapitel beschreiben wir die reale Implementierung der Matrixklassen in LiDIA . Da eine detaillierte Beschreibung aller Klassen inklusive der angebotenen Operationen den Umfang dieser Arbeit sprengen würde, beschränken wir uns in diesem Kapitel auf die Beschreibung der Funktion der einzelnen Klassen innerhalb des Gesamtkonzepts. Für weiterführende Informationen verweisen wir auf [Gro99].

Die aktuelle Implementierung der Matrizen umfaßt 33 Klassen. Diese haben wir, der besseren Anschaulichkeit wegen, in die folgenden drei Implementierungsebenen eingeteilt:

1. **Funktionalitätsebene**
2. **Algorithmenebene**
3. **Repräsentationsebene**

Eine Ebene faßt dabei Klassen ähnlicher Funktionalität, d.h. Klassen, die innerhalb der beschriebenen Gesamtstruktur die gleiche logische Funktion erfüllen, zusammen. In den folgenden Abschnitten stellen wir die Ebenen und die Klassen, die diese beinhalten, detailliert vor.

8.1 Funktionalitätsebene

Die Funktionalitätsebene beinhaltet alle Klassen der Matriximplementierung, die Funktionen entweder dem Benutzer oder abgeleiteten Klassen (z.B. `lattice`–Klassen) zur Verfügung stellen.

8.1.1 Vererbungshierarchie

Den Klassen der Funktionalitätsebene der aktuellen Matriximplementierung liegt folgende Vererbungshierarchie zugrunde.

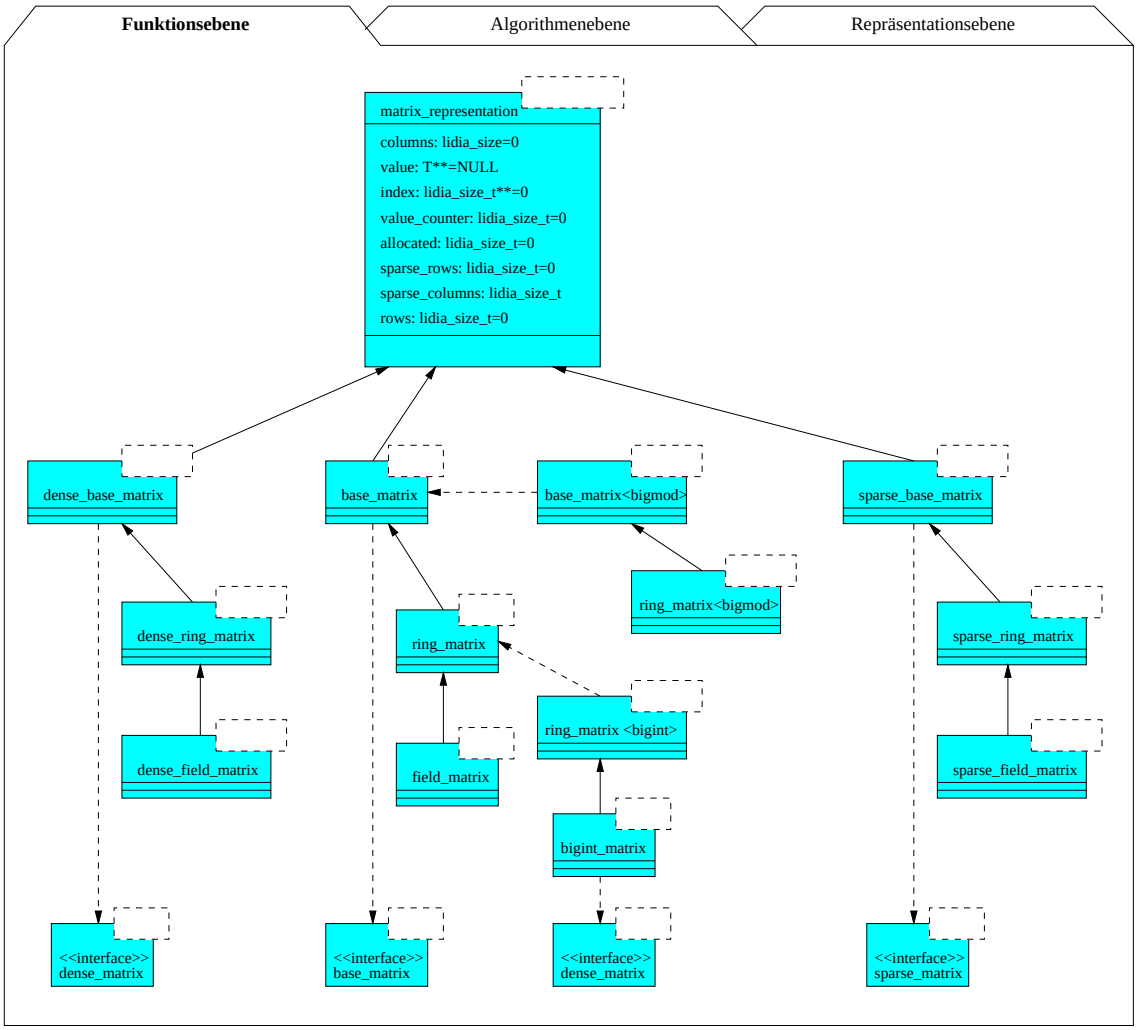


Abbildung 8.1: Funktionsebene

Auf der Basis einer Datenstruktur (`matrix_representation`) unterstützt die aktuelle Implementierung Matrizen in drei Abstraktionsebenen (`base_matrix`, `ring_matrix`, `field_matrix`) und vier Repräsentationen (`dichtbesetzt--zeilenorientiert`, `dichtbesetzt--spaltenorientiert`, `dünnbesetzt--zeilenorientiert`, `dünnbesetzt--spaltenorientiert`). Desweiteren bietet sie dem Nutzer neben einer Reihe von Spezialisierungen zwei statische Interface–Klassen (`dense_matrix`, `sparse_matrix`) und eine dynamische Interface–Klasse (`matrix`) zur Nutzung an.

8.1.2 Datenstruktur

Die einer Matrix zugrundeliegende Datenstruktur wird in der parametrisierten C++ Klasse `matrix_representation` festgelegt. Sie besteht aus sieben Teilen:

1. **Dimensionen der Matrix**

   ```
   lidia_size_t rows
   lidia_size_t columns
   ```

   Diese beiden Elemente dienen der Speicherung der Dimensionen der Matrix.

2. **Startspalte bzw. Startzeile der dünnbesetzten Struktur**

   ```
   lidia_size_t sparse_rows;
   lidia_size_t sparse_columns;
   ```

   Mittels dieser beiden Datenstrukturelemente ist es möglich, Mischrepräsentationen zu verwalten. Die Werte dieser Elemente dienen dabei der Bestimmung des ersten Elementes in dünnbesetzter Repräsentation, d.h. alle Elemente mit einem Zeilenindex größer als `sparse_rows` oder einem Spaltenindex größer als `sparse_columns` werden in dünnbesetzter Repräsentation gespeichert.

3. **Wertearray**

   ```
   T **value;
   ```

   Die Verwaltung der Einträge einer Matrix erfolgt dynamisch. Dieser Pointer zeigt auf das zweidimensionale Array der Matrixeinträge.

4. **Indexarray**

   ```
   lidia_size_t **index;
   ```

   Zur Erzeugung einer dünnbesetzten Repräsentation ist es notwendig, die Indizes der Elemente, d.h. die Position der Elemente innerhalb der Matrix abzuspeichern, die nicht dem Nullelement entsprechen. Über diesen Pointer wird im Bedarfsfall ein zweidimensionales, dynamisch erzeugtes Array adressiert, welches diese Aufgabe übernimmt.

5. **Anzahl der Elemente pro Zeile / Spalte**

   ```
   lidia_size_t *value_counter;
   lidia_size_t *allocated;
   ```

   Als Ergänzung zu dem oben genannten Indexarray ist es notwendig, zur Realisierung einer dünnbesetzten Repräsentation die Anzahl der Elemente pro Zeile bzw. Spalte und die Anzahl der allokierten Elemente pro Zeile bzw. Spalte zu speichern. Diese beiden Array übernehmen diese Aufgabe und sind mit dem Index der jeweiligen Zeile bzw. Spalte indiziert.

6. **Nullelement**

   ```
   T Zero;
   ```

   Dieses Datenstrukturelement dient der Ablage des Nullelementes, d.h. des Elementes, welches aus der dünnbesetzten Repräsentation im Vergleich zur dichtbesetzten Repräsentation entfernt wird. Nicht jeder Datentyp, der als Template–Argument für die Matrixklassen verwendet werden kann, besitzt ein ausgezeichnetes Element bzw. eine ausgezeichnete Funktion zur Bestimmung des Nullelementes dieses Datentyps. Aus diesem Grund ist es erforderlich, das Nullelement als Element der Datenstruktur zu hinterlegen.

7. **Informationen über Struktur, Repräsentation, Orientierung**

   ```
   matrix_flags bitfield;
   ```

   Die Klasse `matrix_flags`, auf die wir im Detail noch eingehen werden, dient dazu, Zusatzinformationen kodiert pro Matrix abzulegen und zu weiteren Berechnungen heranzuziehen.

8.1. Bemerkung [LiDIA]
Der LiDIA–Datentyp `lidia_size_t` kapselt einen Typ, der zur Numerierung verwendet wird. In der aktuellen Implementierung ist dies ein `typdef` auf `long`.

Diese komplexe Datenstruktur ist als kleinstes gemeinsames Vielfaches von drei getrennten Datenstrukturen zu verstehen, wobei die erste zur Speicherung einer dichtbesetzten Matrix, die zweite zur Speicherung einer dünnbesetzten Matrix und die dritte zur Speicherung einer Matrix in gemischter Repräsentation dient. Die Auswahl der einzelnen Datenstrukturen wurde von dem Wunsch geleitet, durch die Vereinigung und Verschmelzung der Einzeldatenstrukturen zu einer Datenstruktur einen nahtlosen Übergang zwischen den Repräsentationen zu ermöglichen. Dies entspricht der Designentscheidung der *Trennung von Datenstruktur und Repräsentation.*

Im folgenden illustrieren wir anhand der Beispielmatrix

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 & 3 \\ 4 & 5 & 6 & 0 & 0 \\ 0 & 7 & 8 & 0 & 9 \\ 0 & 0 & 0 & 10 & 0 \\ 11 & 0 & 0 & 0 & 12 \end{pmatrix} \tag{8.1}$$

in welcher Art und Weise die Datenstruktur zur Realisierung einer dichtbesetzten, dünnbesetzten bzw. gemischten Repräsentation interpretiert wird.

1. **Dichtbesetzte Repräsentation**
   Zur Realisierung der dichtbesetzten Repräsentation werden die Datenstrukturelemente wie folgt belegt:

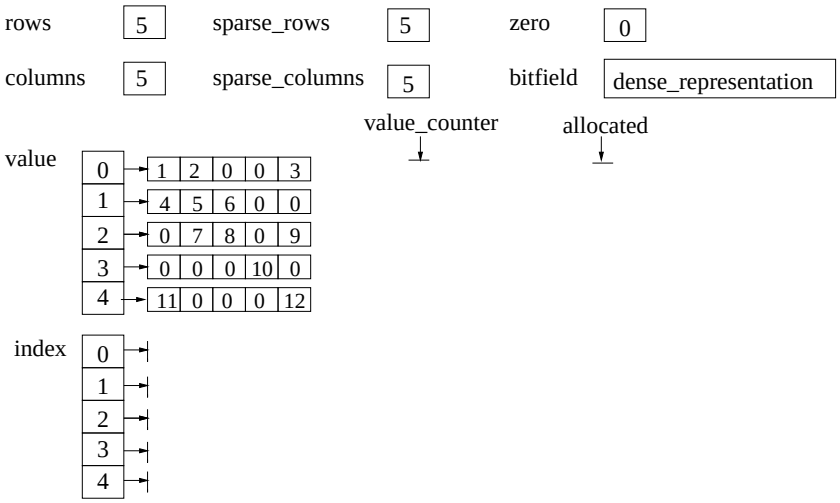


Abbildung 8.2: Dichtbesetzte Repräsentation

Es ist offensichtlich, daß die Datenstrukturelemente `index`, `sparse_rows`, `sparse_columns` `value_counter` und `allocated` in der dichtbesetzten Repräsentation lediglich eine untergeordnete Rolle spielen.

2. **Dünnbesetzte Repräsentation**
   Zur Realisierung einer dünnbesetzten Matrixrepräsentation sind eine Vielzahl von Datenstrukturen bekannt. In der aktuellen Implementierung wird die obige Matrix in dünnbesetzter Darstellung wie folgt abgelegt.

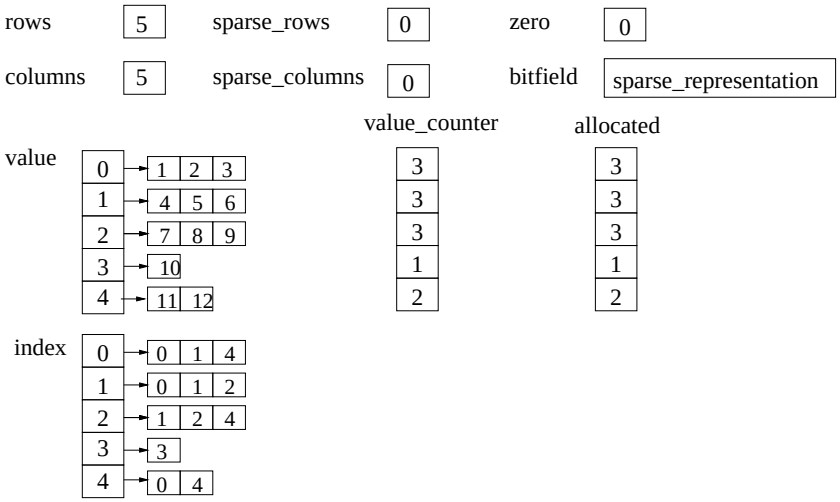


Abbildung 8.3: Dünnbesetzte Repräsentation

Die von uns gewählte Datenstruktur und Repräsentation biete eine hohe Speicherplatzeffizienz in Verbindung mit der Möglichkeit eines nahtlosen Übergangs von der dichtbesetzten zur dünnbesetzten Repräsentation. Als Zwischenstufe taucht dabei die gemischte Repräsentation auf, die im folgenden noch beschrieben wird.

3. **Gemischte Repräsentation**
   Die folgende Datenstrukturbelegung zeigt die obige Beispielmatrix in gemischter

Repräsentation. Die obere linke (3×3)–Teilmatrix wird in dichtbesetzter Repräsentation, während der Rest in dünnbesetzter Repräsentation abgelegt wird.

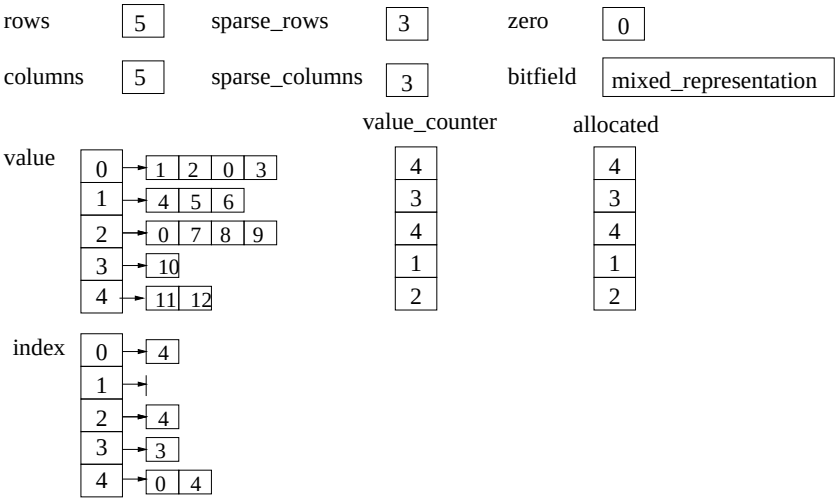


Abbildung 8.4: Gemischte Repräsentation

Zur Vervollständigen der Beschreibung der Datenstruktur einer Matrix schauen wir uns nun den Aufbau der Klasse `matrix_flags` an.

- **Bedeutung:**
  Die Klasse `matrix_flags` dient zur Speicherung von Informationen, die die Arbeit mit Matrizen erleichtern und näher spezifizieren sollen.

- **Datenstruktur:**
  Die Datenstruktur der Klasse besteht aus fünf `unsigned long` Attributen. Jedes Bit eines Attributs wird als Flag, d.h. als Schalter bzgl. einer Information aufgefaßt.

  1. **Informationen über den Ausgabemodus**

     `unsigned long print_mode`

     Die Ein- bzw. Ausgabefunktionen der Matrixklassen sind in der Lage, Matrixdaten unterschiedlicher Formate zu verarbeiten. Dies ist von besonderer Bedeutung im Hinblick auf die Zusammenarbeit mit anderen Computer–Algebra–Systemen, wie z.B. Magma [Mag99], Mathematica [Mat99], Maple [Map99], GP [BBCO99] etc. In diesen Flags wird das Ein- und Ausgabeverhalten der jeweiligen Matrixinstanz festgelegt.

  2. **Informationen über die Repräsentation und Orientierung**

     `unsigned long storage_mode`

     Wie bereits an verschiedenen Stellen betont, kann die Matrixdatenstruktur in unterschiedlicher Art und Weise interpretiert werden. In diesen Flags wird kodiert, welche Repräsentation für die entsprechende Matrix verwendet wird.

  3. **Informationen über die Verteilung der Einträge**

     `unsigned long structure_mode`

Diese Flags dienen der Speicherung von Informationen über die Verteilung der Einträge, die nicht dem Nullelement entsprechen. Mittels dieser Informationen können Spezialalgorithmen angesteuert werden, die diese Zusatzinformation ausnutzen.

4. **Weiterführende Informationen**

   `unsigned long info_mode`

   Diese Flags kodieren Zusatzinformationen zu der in den `structure_mode` Flags kodierten Information.

5. **Informationen für die abgeleiteten Gitterklassen**

   `unsigned long lattice_mode`

   Diese Flags werden von den auf den Matrixklassen aufbauenden Gitterklassen verwendet. Eine detailierte Beschreibung der Bedeutung dieser Flags findet sich in [Wet98, Bac98].

- **Methoden:**
  Diese Klasse stellt neben Konstruktoren und einem Destruktor lediglich Funktionen zum Setzen, Löschen und Abfragen der einzelnen Flags bereit.

8.1.3 Hierarchieklassen

In der aktuellen Implementierung unterscheiden wir Matrixklassen auf drei Ebenen gemäß der Eigenschaften des zur Instantiierung verwendeten Argumenttyps.

1. **Klasse:** `base_matrix`

   - **Bedeutung:**
     Diese Klasse interpretiert eine Matrix als ein Anordnungschema für Elemente einer zugrundeliegenden Menge. Jeder Typ, der einen Zuweisungsoperator (=) und einen Gleichheitsoperator (==) besitzt, kann zur Instantiierung dieser Klasse verwendet werden.
   - **Methoden:**
     Die Funktionen dieser Klasse gliedern sich in die folgenden Bereiche:
     (a) Konstruktoren und Destruktor
     (b) Ein- und Ausgabeoperatoren
     (c) Zugriffsfunktionen
     (d) Insert- und Removefunktionen
     (e) Strukturfunktionen
     (f) Exchange und Swapfunktionen
     (g) Split- und Composefunktionen
     (h) Zuweisungsoperatoren und -funktionen
     (i) Diagonalfunktion
     (j) Bildung der Transponierten

(k) Stream handling Funktionen
(l) Boolesche Funktionen

2. **Klasse:** `ring_matrix`

- **Bedeutung:**
  Diese Klasse setzt voraus, daß die zur Instantiierung verwendete Template–Argumentklasse die Operationen und Eigenschaften eines Rings aufweist. Sie nutzt diese Eigenschaften dazu, entsprechend induzierte Matrixoperationen bereitzustellen. Aufgrund der Vererbungshierarchie stehen ebenfalls die Funktionen und Operationen der Klasse `base_matrix` zur Verfügung.
- **Methoden:**
  Der Funktionsumfang dieser Klasse setzt sich aus folgenden Gruppen zusammen.
  (a) Konstruktoren und Destruktor
  (b) Arithmetische Operatoren und Prozeduren
  (Addition, Subtraktion, Multiplikation, Negation)
  (c) Vergleichsoperatoren
  (d) Trace-Funktion

3. **Klasse:** `field_matrix`

- **Bedeutung:**
  Die Klasse `field_matrix` ist von der Klasse `ring_matrix` abgeleitet und setzt voraus, daß der zur Instantiierung verwendetet Template–Argumenttyp die Operationen und Eigenschaften eines Körpers aufweist. Ermöglicht durch die zusätzlichen Eigenschaften und Operationen eines Körpers werden weitere Funktionen und Operationen zur Verfügung gestellt.
- **Methoden:**
  Die Methoden dieser Template–Klasse gliedern sich in die folgenden Gruppen:
  (a) Konstruktoren und Destruktoren
  (b) Arithmetische Operatoren und Prozeduren

8.1.4 Spezialisierungen

Spezialisierungen für Template–Klassen sind immer dann sinnvoll, wenn der spezielle Argumenttyp entweder neben den in der Template–Klasse definierten Methoden die Definition weiterer Argumenttyp–abhängiger Methoden ermöglicht oder unter Ausnutzung spezieller Eigenschaften des Argumenttyps eine Effizienzsteigerung erreicht.

In der aktuellen Implementierung sind folgende Spezialisierungsklassen enthalten.

1. **Klasse:** `bigint_matrix`

- **Bedeutung:**
  Die Klasse `bigint_matrix` ist eine Spezialisierung der Klasse `ring_matrix` für den Typ `bigint` und stellt Funktionalitäten der Linearen Algebra über $\mathbb{Z}$ bereit.

- **Methoden:**
  Die zusätzlichen Funktionen dieser Klasse werden in die folgenden Kategorien unterteilt:
  (a) Konstruktoren und Destruktor
  (b) Cast-Operatoren
  (c) Pseudo-Division
  (d) Modulooperator
  (e) Normen und Schranken
  (f) Zufallsmatrizen
  (g) Funktionen der Linearen Algebra über $\mathbb{Z}$
    i. Rang, linear unabhängige Zeilen, linear unabhängige Spalten
    ii. Reguläre Ergänzung
    iii. Adjungierte Matrix
    iv. Vielfaches der Gitterdeterminante
    v. Determinante
    vi. Charakteristisches Polynom
  (h) Matrixnormalformen
    i. Hermite–Normalform
    ii. Kern
    iii. Reguläres Urbild
    iv. Bild
    v. Urbild
    vi. Lösen von linearen Gleichungssystemen
    vii. Smith–Normalform
  (i) Basisergänzung
  (j) Konditionierung
  (k) Gaußelimination
  (l) Berechnung des größten, gemeinsamen Teilers

2. **Klasse:** `bigfloat_matrix`

- **Bedeutung:**
  Diese Klasse ist eine Spezialisierung der Klasse `field_matrix` für den Typ `bigfloat`. Als Ergänzung zu den Funktionen der Klasse `field_matrix` stellt diese Klasse Operationen bereit, die bei der Berechnung kürzester Gittervektoren benötigt werden.
- **Methoden:**
  Die Funktionen dieser Klasse gliedern sich in:
  (a) Konstruktoren und Destruktor
  (b) Zufallsmatrizen
  (c) Boolesche Operatoren und Funktionen
  (d) Invertierung
  (e) Gauß–Jordan–Elimination
  (f) Cholesky-Zerlegung
  (g) LR-Zerlegung

(h) QR-Zerlegung

3. **Klasse:** `fp_matrix`

   - **Bedeutung:**
     Aus Gründen der Speicherplatzoptimierung ist es sinnvoll, Matrizen über endlichen Primkörpern als Spezialisierung zu implementieren. Auf Details dieser Spezialisierung gehen wir an dieser Stelle nicht ein, sondern verweisen auf [Gro99]. Die Klasse `fp_matrix` ist eine Spezialisierung der Klasse `field_matrix` für endliche Körper.
   - **Methoden:**
     Diese Klasse stellt die folgenden Funktionen bereit:
     (a) Konstruktoren und Destruktor
     (b) Spaltenstufenform
     (c) Rang, linear unabhängige Zeilen, linear unabhängige Spalten
     (d) Adjungierte Matrix
     (e) Determinante
     (f) Hessenbergform
     (g) Charakteristisches Polynom

8.1.5 Interfaceklassen

Interface–Klassen dienen der Erhöhung der Benutzbarkeit einer Klassenhierarchie, indem sie dem Anwender eine einheitliche Schnittstelle zu allen Instanzen verschiedener Klassen inklusive Spezialisierungen zur Verfügung stellen. Insbesondere entbinden sie ihn von der Notwendigkeit, die Vererbungsstruktur als Ganzes kennen zu müssen.

In der aktuellen Implementierung bieten wir drei Interface–Klassen an:

1. **Interfaceklasse:** `dense_matrix`
   Unter diesem statischen Interface werden in Abhängigkeit von der jeweiligen Template-Argumentklasse Funktionen angeboten, die auf einer zeilenorientierten, dichtbesetzten Repräsentation basieren.

   8.2. Beispiel
   `dense_matrix < char >` entspricht `dense_base_matrix < char >`.
   `dense_matrix < bigint >` entspricht `dense_ring_matrix < bigint >`.

2. **Interfaceklasse:** `sparse_matrix`
   Dieses statische Interface kapselt die zeilenorientierte, dünnbesetzte Repräsentation. In Abhängigkeit von dem jeweiligen Template-Argumenttyp werden die zugehörigen Funktionen zur Verfügung gestellt.

   8.3. Beispiel
   `sparse_matrix < char >` entspricht `sparse_base_matrix < char >`.
   `sparse_matrix < bigint >` entspricht `sparse_ring_matrix < bigint >`.

3. **Interfaceklasse:** `matrix`
   Diese Klasse stellt ein dynamische Interface dar. Es erlaubt die Benutzung aller angebotenen Repräsentationen und bietet wie die statischen Interface-Klassen Funktionalitäten in Abhängigkeit von der jeweiligen Template-Argumentklasse an.

8.4. Bemerkung
Ein weiterer Vorteil der Benutzung von Interfaceklassen liegt darin, daß bereits zur Übersetzungszeit festgestellt wird, ob eine gewünschte Objektinstanz die angeforderte Funktionalität bietet oder nicht. In einem Matrixdesign mit virtuellen Funktionen ist eine solche Information erst zur Laufzeit verfügbar.

8.2 Algorithmenebene

In diesem Abschnitt gehen wir auf die Klassen der Algorithmenebene unserer Matriximplementierung ein. Die Klassen dieser Ebene beinhalten die Kernalgorithmen unserer Implementierung, d.h. sie umfassen auf der einen Seite die repräsentationsunabhängigen Teile der implementierten Algorithmen und auf der anderen Seite die Teile der implementierten Algorithmen, die, über verschiedene Varianten hinweg betrachtet, konstant bleiben.

Desweiteren beinhaltet diese Ebene alle Konfigurationsklassen. Diese Klassen werden wie Algorithmenklassen verwendet, dienen aber lediglich der Vereinfachung des Instantiierungsprozesses.

8.2.1 Vererbungshierarchie

Die Vererbungshierarchie der Klassen dieser Ebene ist in der folgenden Abbildung dargestellt.

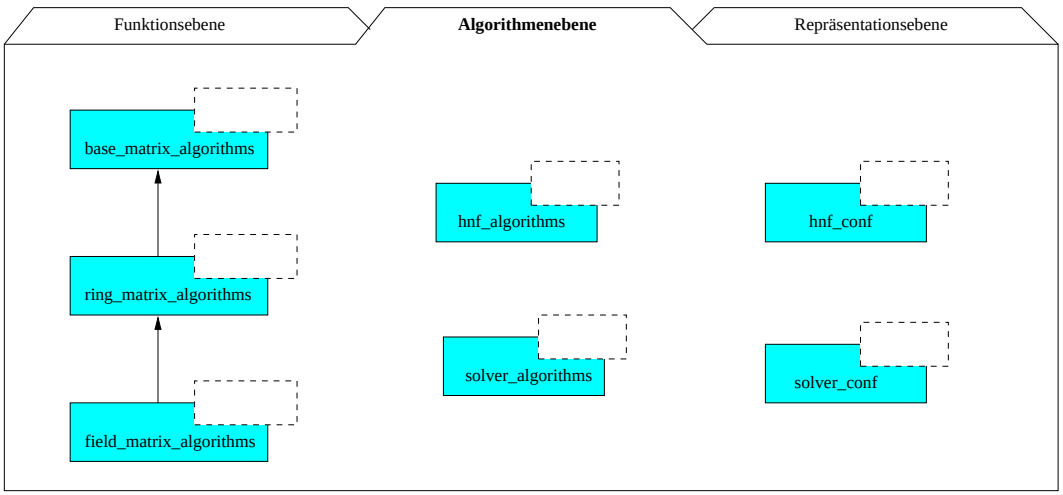


Abbildung 8.5: Algorithmenebene

8.2.2 Algorithmenklassen

Algorithmenklassen sind parametrisierte C++ Klassen. Sie umfassen die ihrem Namen und somit einerseits der Vererbungsstufe und andererseits der Funktion entsprechenden Kernalgorithmen. Ihre Existenz resultiert aus der verwendeten Implementierungstechnik. Bisher sind in LiDIA fünf Algorithmenklassen enthalten:

1. **Klasse:** `base_matrix_algorithms`
   Diese Algorithmenklasse beinhaltet die Kernalgorithmen der Vererbungsstufe `base_matrix`. Dazu gehören der Kernalgorithmus zur Transponiertenbildung, der Kernalgorithmus zur Zerlegung einer Matrix in Teilmatrizen, usw.

2. **Klasse:** `ring_matrix_algorithms`
   Diese Algorithmenklasse beinhaltet die Kernalgorithmen der Vererbungsstufe `ring_matrix`. Dazu gehören die Kernalgorithmen der Matrixaddition, Matrixsubtraktion, usw.

3. **Klasse:** `field_matrix_algorithms`
   Diese Algorithmenklasse beinhaltet die Kernalgorithmen der Vererbungsstufe `field_matrix`. Dazu gehören die Algorithmen zur Berechnung der inversen Matrix, zur Berechnung der Determinanten, usw.

4. **Klasse:** `hnf_algorithms`
   Diese Algorithmenklasse beinhaltet verschiedene Kernalgorithmen, die zur Erzeugung von Algorithmen zur Berechnung der Hermite–Normalform benötigt werden. Die Beschreibung dieser Kernalgorithmen ist Gegenstand eines späteren Teils dieser Arbeit. Im ihm werden wir im Detail sehen, welche Kernalgorithmen und welche Modulalgorithmen dem Anwender zur Verfügung stehen.

5. **Klasse:** `solver_algorithms`
   Diese Algorithmenklasse dient der Realisierung von Gleichungssystemlösern. In ihr sind die Kernbestandteile des Lanczos-, des Wiedemann- und des Konjugierten Gradienten–Algorithmus enthalten.

8.2.3 Konfigurationsklassen

Die Implementierungstechnik *Template–Kernel / Template–Modul* führt bei intensiver und iterativer Verwendung zu komplizierten Instantiierungsanweisungen. Konfigurationsklassen dienen dazu, diese komplizierten Anweisungen aufzubrechen. Sie fassen Template–Parametertypenlisten in sinnvollen Konfigurationen zusammen und erleichtern somit erheblich den Instantiierungsprozeß und die Lesbarkeit des Sourcecodes. Die Vergrößerung der Codebasis durch diese Klassen ist vernachlässigbar und gefährdet somit nicht die Wartbarkeit des Gesamtsystems.

In der aktuellen Implementierung sind bisher zwei Konfigurationsklassen enthalten.

1. **Klasse:** `hnf_conf`
   Diese Klasse enthält alle Konfigurationen zur Instantiierung von Funktionen zur Berechnung der HNF.

2. **Klasse:** `solver_conf`
   Diese Klasse umfaßt die Konfigurationen zur Instantiierung der in LiDIA enthaltenen Gleichungssystemlöser.

8.3 Repräsentationsebene

Das Repräsentationslevel umfaßt alle Klassen, die Basisroutinen bereitstellen, die die einer Matrix zugrundeliegende Datenstruktur in einer festgelegten Art und Weise interpretieren und somit als Module für die Kernalgorithmen dienen können. Dieses Level ist in der folgenden Abbildung dargestellt.

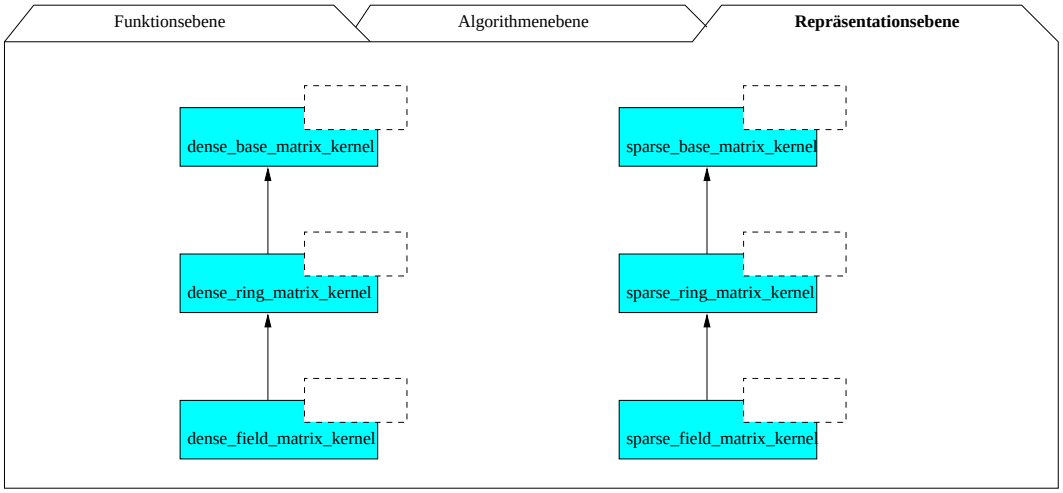


Abbildung 8.6: Repräsentationsebene

8.3.1 Repräsentationsklassen

Wie bereits in den vorangegangenen Kapiteln erwähnt, besteht eine Designentscheidung darin, eine Basisdatenstruktur in unterschiedlicher Art und Weise zu interpretieren und so verschiedene Repräsentationsarten zu implementieren. In der aktuellen Implementierung unterstützen wir vier Repräsentationen, die sich aus der Kombination von Zeilen- und Spaltenorientierung sowie dichtbesetzter und dünnbesetzter Repräsentation ergeben.

Jede der folgenden Klassen enthält Routinen, die zum Umgang mit einer Matrix, die gemäß der dem Namen der Klasse entsprechenden Repräsentation abgelegt ist, benötigt werden.

1. **Klasse:** `dense_row_oriented_rep`
2. **Klasse:** `dense_column_oriented_rep`
3. **Klasse:** `sparse_row_oriented_rep`
4. **Klasse:** `sparse_column_oriented_rep`

Teil IV

Framework II: Algorithmen zur Berechnung der Hermite–Normalform

Kapitel 9

Gliederung

In diesem Teil der Arbeit stellen wir eine systematische Einteilung der uns bekannten Algorithmen zur Berechnung der Hermite–Normalform vor. Wir erläutern die Zusammenhänge zwischen verschiedenen Algorithmen, bilden Algorithmenklassen und untersuchen das Laufzeitverhalten dieser Algorithmen in Bezug auf die Entwicklung der Eintragsdichte, die Entwicklung der Eintragsgröße der entstehenden Zwischenmatrizen, der Effizienz und somit im Hinblick auf ihre Anwendbarkeit.

Dabei verzichten wir auf eine detaillierte Beschreibung der Implementierung der jeweiligen HNF–Algorithmen, da dies den Umfang dieser Arbeit bei weitem übersteigen würde. Die verwendeten Implementierungsideen wurden alle im vorangegangenen Teil dieser Arbeit beschrieben.

Zur Einteilung der Algorithmen verwenden wir die folgende zweistufige Gliederung:

- Auf der ersten Ebene unterscheiden wir zwischen modularen Algorithmen, d.h. Algorithmen, die zur Beschränkung der Größe der Zwischeneinträge die Kenntnis eines Vielfachen der Gitterdeterminanten ausnutzen, und nichtmodularen Algorithmen.
- Auf der zweiten Ebene unterscheiden wir die Algorithmen gemäß der Eliminationsstrategie, d.h. gemäß der Reihenfolge, in der die Nicht–Nulleinträge während der Berechnung der HNF eliminiert werden.

Viele aus der Literatur bekannte Algorithmen verwenden die gleiche Eliminationsstrategie und sind somit vom Grundalgorithmus, d.h. von der Verarbeitungsstrategie her ähnlich. Die Unterschiede konzentrieren sich auf wenige Algorithmusteile. Deshalb bietet es sich an, gleichbleibende, konstante Algorithmusteile von solchen, die sich ändern und somit den Unterschied zwischen zwei Algorithmen darstellen, zu trennen. Die variablen Teile nennen wir im weiteren Verlauf dieser Arbeit **Modulalgorithmen**, während wir den konstanten Teil eines Algorithmus als den **Kernalgorithmus** bezeichnen.

Diese Unterscheidung ermöglicht es uns, präzise die Unterschiede zwischen ähnlichen Algorithmen herauszuarbeiten und auf Verbesserungen bzw. eigene Forschungsgebiete hinzuweisen.

9.1. Bemerkung

Wir haben an dieser Stelle die gleichen Begriffe gewählt, die bereits im vorherigen Teil dieser Arbeit in Bezug auf eine spezielle Implementierungstechnik eingeführt wurden. Dies soll hervorheben, wie die Variantenvielfalt dieser Algorithmen in die Matrixklassen der Bibliothek LiDIA integriert wurde.

Kapitel 10

Nichtmodulare Algorithmen zur Berechnung der HNF

In diesem Kapitel werden Algorithmen zur Berechnung der Hermite–Normalform vorgestellt, die nicht die Kenntnis eines Vielfachen der Gitterdeterminanten voraussetzen. Zur Veranschaulichung stellen wir die Algorithmen für den Fall der Berechnung der HNF von ganzzahligen Matrizen mit vollem Zeilenrang dar. Dies erlaubt es uns, von unwesentlichen Algorithmusteilen, die lediglich der Handhabung von fehlenden Diagonaleinträgen dienen, zu abstrahieren. Da die Modifikationen, die notwendig sind, um Matrizen mit beliebigem Zeilenrang zu handhaben, offensichtlich sind und das Verhalten in Bezug auf die gewählten Kenngrößen nicht beeinflussen, stellt dies keine Einschränkung der vorgestellten Ergebnisse dar.

Wir unterscheiden im folgenden drei Algorithmenklassen:

1. Zeilenweise Berechnung der HNF
2. Spaltenweise Berechnung der HNF
3. Hauptminorenweise Berechnung der HNF

10.1 Zeilenweise Berechnung der HNF

Vorgehensweise:

Die Basis dieser Algorithmenklasse bildet die zeilenweise Elimination. Algorithmen dieser Klasse bewegen sich zeilenweise beginnend bei der letzte Zeile aufwärts und bringen iterativ die Elemente links der aktuellen Arbeitsposition (Diagonalposition) durch die Bildung geeigneter Linearkombinationen auf Null.

Schema:

$$\begin{pmatrix} * & \dots & * & * \\ \vdots & \ddots & \vdots & \vdots \\ * & \dots & * & * \\ * & \dots & * & * \\ * & \dots & * & * \end{pmatrix} \begin{pmatrix} * & \dots & * & * \\ \vdots & \ddots & \vdots & \vdots \\ * & \dots & * & * \\ * & \dots & * & * \\ 0 & \dots & 0 & * \end{pmatrix} \cdots \begin{pmatrix} * & \dots & * & * & * \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ * & \dots & * & * & * \\ 0 & \dots & 0 & * & * \\ 0 & \dots & 0 & 0 & * \end{pmatrix}$$

Aufgrund unterschiedlicher Strategien, Nebendiagonalelemente modulo dem zugehörenden Diagonalelement zu reduzieren, d.h. zu normalisieren, unterscheidet man die folgenden zwei Kernalgorithmusvarianten.

Kernalgorithmen:

10.1. Algorithmus

Kernalgorithmus zur zeilenweisen HNF–Berechnung mit Zeilennormalisierung: HNF_{Z1}

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, $\mathrm{rank}_{\mathbb{Z}}(A) = m$
AUSGABE: $\mathrm{HNF}(A)$

[Initialisierung]
(1) $i = m-1; j = n-1; l = 0;$
(2) **while** (i $\geq 0 \wedge$ j ≥ 0) **do**
[Zeilenweise Elimination]
(3) $A' = \begin{pmatrix} a_{0,0} & \dots & a_{0,j} \\ \vdots & \ddots & \vdots \\ a_{i,0} & \dots & a_{i,j} \end{pmatrix};$
(4) $T = \mathbf{mgcd}(\mathbf{A}');$ $\quad //a'_{i,*} \cdot T = (0, \dots, 0, \gcd(a'_{i,*}))$
(5) $A = A \cdot \begin{pmatrix} T & 0 \\ 0 & I_l \end{pmatrix};$
[Zeilenweise Normalisierung]
(6) $A = \mathbf{normalize\_row}(\mathbf{A}, \mathbf{i}, \mathbf{j});$
(7) i– –; j– –; l++;
(8) **od**
(9) **return** (A);

Bei diesem Algorithmus werden Nebendiagonalelemente durch die Funktion *normalize_row* zum frühestmöglichen Zeitpunkt normalisiert. Die Reihenfolge, in der die Elemente rechts des (aktuellen) Diagonalelementes reduziert werden, hat keinen Einfluß auf die Kenngrößen, anhand derer in dieser Arbeit HNF–Algorithmen beurteilt werden. Aus diesem Grund betrachten wir diese Funktion nicht als Modulalgorithmus, sondern als zum Kernalgorithmus gehörig.

10.2. Algorithmus

Zeilennormalisierung: *normalize_row*

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, Zeilenindex i, Spaltenindex j
AUSGABE: A mit $a_{i,j} > a_{i,j+1}, \ldots, a_{i,n-1} > 0$.

[Zeilennormalisierung]
(1) **for** $(k = j+1; k < n; k++)$ **do**
(2) $pos\_div\_rem(a_{i,k}, a_{i,j}, q, r);$ $//a_{i,k} = q \cdot a_{i,j} + r,\ a_{i,j} \geq r \geq 0$
(3) $a_{*,k} = a_{*,k} - q \cdot a_{*,j};$
(4) **od**
(5) **return** (A);

Somit verbleibt in diesem Kernalgorithmus lediglich die **mgcd**–Routine als Modulalgorithmus.

10.3. Bemerkung [LiDIA]
Die Methode $pos\_div\_rem(a, b, c, d)$ dividiert a mit Rest durch b, so daß gilt $a = c \cdot b + d$ und $0 \leq d < b$.

Demgegenüber steht ein Kernalgorithmus, bei dem in einer zusätzlichen Phase am Ende des Algorithmus die Nebendiagonalelemente modulo dem entsprechenden Diagonalelement reduziert werden. Diese Phase wird im folgenden durch den Funktionsnamen *normalize_matrix* symbolisiert.

10.4. Algorithmus

Kernalgorithmus zur zeilenweisen HNF–Berechnung mit Matrixnormalisierung: HNF_{Z2}

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, $\mathrm{rank}_{\mathbb{Z}}(A) = m$
AUSGABE: $\mathrm{HNF}(A)$

[Initialisierung]
(1) $i = m-1; j = n-1; l = 0;$
(2) **while** $(\mathrm{i} \geq 0 \wedge \mathrm{j} \geq 0)$ **do**
[Zeilenweise Elimination]
(3) $A' = \begin{pmatrix} a_{0,0} & \ldots & a_{0,j} \\ \vdots & \ddots & \vdots \\ a_{i,0} & \ldots & a_{i,j} \end{pmatrix};$
(4) $T = \mathbf{mgcd}(\mathbf{A}');$ $//a'_{i,*} \cdot T = (0, \ldots, 0, \gcd(a'_{i,*}))$

(5) $A = A \cdot \begin{pmatrix} T & 0 \\ 0 & I_l \end{pmatrix}$;
(6) i– –; j– –; l++;
(7) **od**
[Matrixnormalisierung]
(8) $A =$ **normalize_matrix**(**A**, **i**, **j**);
(9) **return** (A);

Die Reihenfolge, in der die Nebendiagonalelemente der oberen Dreiecksmatrix, die als Ergebnis der Eliminationsphase entsteht, reduziert werden, hat Einfluß auf die Kenngrößen. Somit variieren die unterschiedlichen Algorithmen gemäß der Auswahl des Matrixnormalisierungsalgorithmus.

In diesem Kernalgorithmus werden folglich die beiden Modulalgorithmen **mgcd** und **normalize_matrix** verwendet.

Nachdem wir die Kernalgorithmen vorgestellt haben, gehen wir im folgenden auf die Modulalgorithmen ein, die in Verbindung mit den Kernalgorithmen vollständige HNF–Algorithmen repräsentieren.

10.1.1 mgcd–Berechnung

Der erste Modulalgorithmus, der die obigen Kernalgorithmen zu HNF–Algorithmen komplettiert, berechnet den größten gemeinsamen Teiler g einer Menge von Zahlen $v = (v_0, \ldots, v_{n-1})$ (mgcd = multiple greatest common divisor) inklusive einer unimodularen Transformationsmatrix T, so daß gilt $v \cdot T = (0, \ldots, 0, g)$.

Präziser formuliert betrachten wir in diesem Abschnitt die Aufgabe, für die Elemente der letzten Zeile einer ganzzahligen $(m \times n)$–Matrix A den größten gemeinsamen Teiler zu berechnen, diesen durch unimodulare Transformationen auf die Position $(m-1, n-1)$ zu bringen und alle übrigen Elemente in der Berechnungszeile durch unimodulare Transformationen zu entfernen.

In Matrixschreibweise ergibt sich folgende Aufgabenstellung:

Berechne eine unimodulare Matrix T mit

$$A \cdot T = \begin{pmatrix} x_{0,0} & \cdots & x_{0,n-2} & x_{0,n-1} \\ \vdots & \ddots & \vdots & \vdots \\ x_{m-2,0} & \cdots & x_{m-2,n-2} & x_{m-2,n-1} \\ 0 & \cdots & 0 & x_{m-1,n-1} \end{pmatrix}$$

und $x_{m-1,n-1} = \mathrm{ggT}(a_{m-1,0}, \ldots, a_{m-1,n-1})$.

Wie bereits in den Arbeiten [The95, HM94a] und vielen anderen dargestellt wurde, spielt dieser Modulalgorithmus eine bedeutende Rolle bei der Kontrolle des Anwachsens der Größe der Zwischenergebnisse und damit für die Ausführungseffizienz.

Desweiteren ist er von Bedeutung für

1. die Anzahl der neu entstehenden Einträge,
2. die Größe der Einträge in der Transformationsmatrix,
3. die Dichte der Einträge in der Transformationsmatrix und
4. damit für die Größe und Dichte der Zwischeneinträge während der Durchführung der HNF–Berechnung.

10.5. Bemerkung
Wir betrachten mgcd–Algorithmen in Bezug auf ihre Anwendung auf Matrizen, da einige der im folgenden vorgestellten mgcd–Algorithmen spezielle Eigenschaften der Matrix zur Auswahl einer geeigneten Vorgehensweise ausnutzen.

Im folgenden stellen wir Algorithmen vor, die den größten gemeinsamen Teiler einer Menge von ganzen Zahlen inklusive einer entsprechenden Transformationsmatrix berechnen. Für jeden dieser Algorithmen untersuchen wir die in Tabelle 10.1 zusammengestellten Kenngrößen.

Symbol	Bezeichnung
$L_0(v)$	Anzahl der Nicht–Nulleinträge des Eingabevektors v
$\varrho(v)$	Eintragsdichte des Eingabevektors v
$L_\infty(v)$	maximaler Eintrag des Eingabevektors v
$L_0(x)$	Anzahl der Nicht–Nulleinträge des Darstellungsvektors x
$L_0(T)$	Anzahl der Nicht–Nulleinträge der Transformationsmatrix T
$L_\infty(x)$	maximaler Eintrag des Darstellungsvektors x
$L_\infty(T)$	maximaler Eintrag der Transformationsmatrix T
$\Delta_\infty(T)$	durchschnittliche Eintragsgröße der Transformationsmatrix T
t	Laufzeit in CPU–Sekunden

Tabelle 10.1: Tabelle der betrachteten Kenngrößen

Experiment zum Aufbau der Datenbasis:
Zum Aufbau der Datenbasis führen wir das folgende Experiment durch. Wir wenden den jeweiligen Algorithmus auf ganzzahlige, zufällige (500×500)-Matrizen steigender Eintragsdichte und steigender Eintragsgröße an. Jede Berechnung wiederholen wir 10-mal, protokollieren jeweils die oben genannten Kenngrößen und mitteln die erzielten Ergebnisse über die 10 Iterationen. Detailliert betrachtet, erzeugen wir Matrizen A mit

$$\begin{aligned} \varrho(A) &\in \{100, 90, 80, 70, 60, 50, 40, 30, 20, 10, 5, 2, 1, 0.5, 0.2, 0.1\}, \\ L_\infty(A) &< B \; mit \; B \in \{100, 1000, 10000, 100000\} \\ &und \\ a_{i,j} &\in_R \{-B, \ldots, B\} \; mit \; 0 \leq i, j < 500. \end{aligned}$$

Jeder unterschiedlichen Eintragsdichte entspricht eine Zeile der jeweiligen, der Größe der Einträge entsprechenden Tabelle. Als Matrixrepräsentation wird die dichtbesetzte, zeilenorientierte Repräsentation verwendet.

10.6. Bemerkung
Die Erforschung der Eigenschaften von Algorithmen zur Berechnung des größten gemeinsamen Teilers einer Menge von ganzen Zahlen ist ein Forschungsgebiet an sich. In dieser Arbeit ist die mgcd–Berechnung lediglich ein Baustein zur Komplettierung von Algorithmen zur Berechnung der Hermite–Normalform.

10.1.1.1 mgcd–Algorithmen auf der Basis paarweiser gcd–Berechnung

Naiver Algorithmus
Die einfachsten und historisch ältesten Algorithmen [Her51, Bra70], die eine Transformationsmatrix gemäß der obigen Aufgabenstellung konstruieren, berechnen iterativ den größten gemeinsamen Teiler von jeweils zwei benachbarten Elementen inklusive seiner Darstellung.

$$\gcd(\ldots \gcd(\gcd(v_0, v_1), v_2), \ldots), v_{n-1})$$

Mittels dieser Information konstruieren sie eine unimodulare Transformationsmatrix. Das Produkt aller so konstruierten Matrizen bildet die gesuchte Transformationsmatrix T.

Der Algorithmus verarbeitet die Elemente der letzten Zeile v der Matrix A in der in der folgenden Abbildung dargestellten Reihenfolge. Dabei steht g_{i-j} für $\gcd(v_i, \ldots, v_j)$.

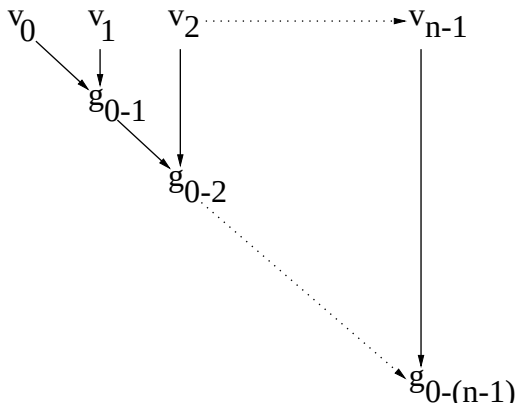


Abbildung 10.1: Verarbeitungsreihenfolge $mgcd_{linear}$

10.7. Bemerkung
Verschiedene Algorithmen, die sich dieser Vorgehensweise bedienen, unterscheiden sich untereinander lediglich darin, wie sie den größten gemeinsamen Teiler (gcd = greatest common divisor) zweier Elemente inklusive seiner Darstellung berechnen. Insbesondere können optimierte Algorithmen verwendet werden [BS96, Web95].

10.8. Algorithmus

Linear iterativer mgcd–Algorithmus: $mgcd_{linear}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$
AUSGABE: $T \in \mathrm{GL}_n(\mathbb{Z})$ mit $a_{m-1,*} \cdot T = (0, \ldots, 0, \gcd(a_{m-1,*}))$.

[Initialisierung]
(1) $v = (a_{m-1,0}, \ldots, a_{m-1,n-1})$;
(2) $T = I_n$;
(3) **for** $(i = n-1; v_i = 0 \wedge\ i \geq 0; i--)$ **do**
(4) **od**
(5) **if** $(i \neq n-1)$ **then**
(6) $swap(v_i, v_{n-1})$;
(7) $T.swap\_columns(i, n-1)$;
(8) **fi**
[mgcd–Algorithmus]
(9) **for** $(i = 1; i < n; i++)$ **do**
(10) g = xgcd(v_{i-1},v_i, a, b);
(11) $\mathrm{U} = \begin{pmatrix} I_{i-1} & & & \\ & -\frac{v_i}{g} & a & \\ & \frac{v_{i-1}}{g} & b & \\ & & & I_{n-i} \end{pmatrix}$;
(12) $v = v \cdot U$;
(13) $T = T \cdot U$;
(14) **od**
(15) **return** (T);

Ergebnisse:
Anhand der praktischen Ergebnisse, die in den Tabellen 10.2, 10.3, 10.4 und 10.5 zu sehen sind, kann man folgende Sachverhalte feststellen:

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	0.04
Bereich (bis)	2.9	1945.2	0.6e16	0.6e21	0.1e21	3.2
abhängig von $L_0(v)$	√	√	√	√	√	√
abhängig von $L_\infty(A)$	√	√	√√	√√	√√	√
abhängig von Eintrags-verteilung von A	–	–	–	–	–	–

1. **Anzahl der Nicht–Nulleinträge des Darstellungsvektors** x **(**$L_0(x)$**)**
Im Durchschnitt kann der größte, gemeinsame Teiler als Linearkombination von 2 oder 3 Einträgen des Eingabevektors dargestellt werden. Es besteht lediglich eine

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2.1	1537.5	34.5	3415.5	736.23	3.066
450.1	90.02	99	2.2	1476.4	17.6	1742.4	410.915	2.764
403.9	80.78	99	2.4	1457.1	47.9	4742.1	994.343	2.481
350.7	70.14	99	2.2	1260.7	31.1	3078.9	716.1	2.159
295.6	59.12	98.9	2.3	1171.2	71.3	7003.6	1263.89	1.826
253.4	50.68	98.8	2.4	1095.6	93.4	9227.9	1271.95	1.55
198.7	39.74	98.7	2.3	951.2	73.1	7203.1	1106.24	1.228
156.3	31.26	98.6	2.2	839.3	45.6	4508.2	568.87	0.964
97	19.4	98.7	2.5	737	105.4	10423.1	1015.43	0.607
51.6	10.32	98.1	2.3	614.8	74.4	7204.9	537.169	0.332
29.5	5.9	97.1	2.2	560.9	40.4	3869.1	150.655	0.205
11	2.2	93.6	2.3	522.3	32.2	3038.2	36.0094	0.099
5.4	1.08	82.9	2.2	508.4	17.5	1068	7.88906	0.068
2.8	0.56	62.7	2	503.9	87.1	929	7.47668	0.054
1.8	0.36	57.3	1.7	501.9	27.7	70.4	1.25503	0.048
0.5	0.1000000	18.8	1	500	1	1	1	0.044

Tabelle 10.2: $mgcd_{linear}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.5	1745.4	411355	0.411e9	0.918e8	3.114
449	89.8	997.6	2.4	1573.7	3892.5	0.389e7	850292	2.795
398.2	79.64	996	2.6	1533.1	4318.4	0.431e7	672883	2.481
348.9	69.78	996.6	2.3	1299.7	1585.9	0.158e7	297680	2.174
303.2	60.64	996.2	2.4	1224.6	11312.8	0.113e8	0.179e7	1.893
243.8	48.76	995.1	2.4	1079.2	1958.2	0.196e7	346711	1.519
197.8	39.56	992.8	2.3	952.3	380.2	375937	65913.4	1.233
149	29.8	992.9	2.5	871.3	207480	0.206e9	0.257e8	0.93
98	19.6	987.6	2.4	731.9	3695	0.359e7	364767	0.615
48.4	9.68	978.9	2.3	608.5	2456.1	0.240e7	114215	0.317
25.7	5.14	962.9	2.4	557.9	3486.9	0.345e7	118587	0.184
8.6	1.72	828.9	2.5	517.9	14330.7	0.103e8	85236.8	0.09
6.2	1.24	887.6	2.4	511.8	4503.1	0.244e7	12764.7	0.075
2.6	0.52	624.3	2.1	503.7	103.7	36733.6	118.865	0.056
2	0.4	567.1	1.7	502.4	651.9	554056	1385.2	0.051
1.1	0.22	257.8	1.3	501	29.9	24087.3	86.2878	0.047

Tabelle 10.3: $mgcd_{linear}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2.7	1846	0.970e7	0.969e11	0.157e11	3.17
455.2	91.04	9985	2.7	1725.3	0.652e8	0.652e12	0.136e12	2.879
397.3	79.46	9961.8	2.4	1448.4	0.125e7	0.125e11	0.252e10	2.511
352.3	70.46	9977.6	2.2	1273.7	0.759e7	0.756e11	0.153e11	2.216
301.7	60.34	9968.3	2.4	1221	604822	0.604e10	0.856e9	1.899
252.5	50.5	9945.4	2.6	1157.6	0.274e8	0.273e12	0.333e11	1.592
197.8	39.56	9954.8	2.8	1048.3	0.172e9	0.170e13	0.211e12	1.252
151.5	30.3	9926	2.4	861.7	109295	0.109e10	0.119e9	0.952
99.9	19.98	9907.7	2.9	782.8	0.115e11	0.114e15	0.136e14	0.632
51.9	10.38	9677.3	2.7	635.4	0.141e9	0.128e13	0.620e11	0.341
26.4	5.28	9739.2	2.2	555.4	175006	0.170e10	0.550e8	0.189
12.8	2.56	9122.1	2.3	526.3	143880	0.131e10	0.130e8	0.111
4.2	0.84	8115.2	2.5	507.5	0.402e7	0.264e11	0.872e8	0.063
3.9	0.78	8317.4	2.2	506.6	496892	0.335e10	0.258e8	0.062
1.7	0.34	4758.2	1.5	502.2	409162	0.141e10	0.405e7	0.054
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.045

Tabelle 10.4: $mgcd_{linear}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.9	1945.2	0.696e16	0.696e21	0.154e21	3.172
452.1	90.42	99823.7	2.5	1626	0.140e9	0.139e14	0.264e13	2.859
399.3	79.86	99763.5	2.5	1494.8	0.147e9	0.146e14	0.310e13	2.527
352.9	70.58	99657.1	2.4	1338.9	0.366e8	0.365e13	0.737e12	2.25
307.4	61.48	99466.2	2.5	1260.6	0.838e11	0.831e16	0.156e16	1.956
251.8	50.36	99431.8	2.2	1051.2	0.477e8	0.473e13	0.970e12	1.589
203.4	40.68	99511.4	2.1	926.1	0.132e7	0.132e12	0.196e11	1.278
150.7	30.14	99297.6	2.6	887.6	0.299e8	0.296e13	0.325e12	0.96
104.3	20.86	98793.9	2.6	763.4	0.202e9	0.196e14	0.152e13	0.661
54.8	10.96	97747.7	2.4	628	0.681e8	0.667e13	0.359e12	0.354
24.6	4.92	96337.4	2.8	568	0.730e12	0.721e17	0.188e16	0.18
11.5	2.3	92767.5	2	521	17939.8	0.165e10	0.269e8	0.107
5.1	1.02	79355.4	2.1	508.4	0.591e7	0.286e11	0.106e9	0.065
2.3	0.46	58436	1.7	503	5079.5	0.213e9	0.107e7	0.053
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.043
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.048

Tabelle 10.5: $mgcd_{linear}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

geringe Abhängigkeit von der Eintragsdichte bzw. der durchschnittlichen Eintragsgröße der Eingabematrix. Die Eintragsverteilung der Eingabematrix wird im mgcd–Algorithmus nicht berücksichtig. Diese Beobachtungen sind auf die zufällige Wahl der Einträge zurückzuführen.

2. **Anzahl der Nicht–Nulleinträge der Transformationsmatrix T ($L_0(T)$)**
   Der Anzahl der zur Darstellung des größten gemeinsamen Teilers benötigten Elemente entsprechend ergibt sich die Anzahl der Nicht–Nulleinträge der Transformationsmatrix T. Beachten muß man dabei, daß, nachdem der größte gemeinsame Teiler gefunden ist, die übrigen, bisher nicht bearbeiteten Elemente der letzten Zeile der Eingabematrix mittels geeigneter Linearkombinationen eliminiert werden.

3. **maximaler Eintrag des Darstellungsvektors x ($L_\infty(x)$)**
   Im Gegensatz zu den beiden vorangehenden Kenngrößen hängt die Größe des maximalen Eintrags des Darstellungsvektors von der Eintragsdichte und der Größe der Einträge der Startmatrix ab. Auffällig ist, daß im allgemeinen der maximale Eintrag des Darstellungsvektors die Schranke $L_\infty(A)$ deutlich übersteigt. Die Differenz steigt überproportional mit steigender Größe der Einträge der Startmatrix an.

4. **maximaler Eintrag der Transformationsmatrix T ($L_\infty(T)$)**
   Die maximalen Einträge der Transformationsmatrix übersteigen deutlich die jeweiligen maximalen Einträge des Darstellungsvektors. Die Abhängigkeiten zwischen der Größe des maximalen Eintrags des Darstellungsvektors und der Eintragsdichte der Startmatrix bzw. der Größe der Einträge der Startmatrix übertragen sich eins zu eins auf den maximalen Eintrag der Transformationsmatrix.

5. **durchschnittliche Eintragsgröße der Transformationsmatrix T ($\Delta_\infty(T)$)**
   Desweiteren ist die durchschnittliche Eintragsgröße der Nicht–Nulleinträge der Transformationsmatrix deutlich über dem maximalen Eintrag des Darstellungsvektors, bleibt aber auch deutlich hinter dem maximalen Eintrag der Transformationsmatrix zurück. Die oben erwähnten Abhängigkeiten übertragen sich wiederum eins zu eins auf die hier betrachtete Kenngröße.

6. **Laufzeit (t)**
   Die Laufzeiten bewegen sich zwischen 0,01 bis 3,5 CPU–Sekunden, wobei eine deutliche Abhängigkeit zur Eintragsdichte zu erkennen ist. Die Anzahl der durchzuführenden Eliminationen ist vorrangig für diese Abhängigkeit verantwortlich. Der Einfluß der Größe der Einträge der Startmatrix auf die Laufzeit ist minimal.

Im Laufe einer HNF–Berechnung wird der oben beschriebene Modulalgorithmus mehrfach, nämlich einmal pro Zeile, ausgeführt. Betrachtet man die Größe der Darstellungskoeffizienten, die Größe der Einträge in der Transformationsmatrix und vor allem die überproportionalen Abhängigkeit dieser Kenngrößen von der Größe der Einträge in der Startmatrix, so wird klar, daß dieser Modulalgorithmus einem überproportionalen Anwachsen der Einträge bei der Durchführung einer HNF–Berechnung Vorschub leistet.

Minimierung der Größe der Darstellungskoeffizienten

Eine erste Idee, das Laufzeitverhalten des obigen Algorithmus zu verbessern, ist die Minimierung der Größe der Koeffizienten der Darstellung des größten gemeinsamen Teilers.

Dabei steht man zunächst vor der Frage, was man unter einer *minimalen* Darstellung zu verstehen hat.

Für die Berechnung des größten gemeinsamen Teilers zweier Zahlen hat sich folgendes Verständnis durchgesetzt. Als minimale Darstellung des größten gemeinsamen Teilers g zweier ganzer Zahlen a_0 und a_1 versteht man die Koeffizienten x_0 und x_1, so daß gilt:

$$\begin{aligned} g = \gcd(a_0, a_1) &= a_0 \cdot x_0 + a_1 \cdot x_1 \\ |x_0| &\leq \frac{|a_1|}{2} \cdot g \\ |x_1| &\leq \frac{|a_0|}{2} \cdot g \end{aligned}$$

Für einen Vektor mit n Elementen ist die Lage nicht so klar. Die Berechnung des größten gemeinsamen Teilers eines Vektors $a = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}^n$ läßt sich als sie Bestimmung eines Vektors $x \in \mathbb{Z}^n$ mit

$$g = \gcd(a) = a^T \cdot x = \sum_{i=0}^{n-1} a_i \cdot x_i$$

interpretieren. Somit ist es sinnvoll, die Minimalität der Darstellung in Abhängigkeit von einer zugrundeliegenden Metrik zu betrachten.

In diesem Abschnitt betrachten wir die Minimalität in Bezug auf die L_∞–Norm. Für diese Norm ist der Vektor unter den möglichen Lösungsvektoren minimal, der die betragsmäßig kleinsten Elemente enthält.

10.9. Bemerkung

- Es wäre wünschenswert, daß die obere Schranke der Einträge in der Darstellung des gcd mit steigender Anzahl an Elementen kleiner wird. Dies ist jedoch nicht möglich, wie man leicht anhand der folgenden Menge ganzer Zahlen verifizieren kann. Betrachtet man die Menge $\{2, 2 \cdot k + 1, 2 \cdot k + 1, \ldots, 2 \cdot k + 1\}$ mit $k \geq 1$, dann bleibt die obere Schranke in der Darstellung des größten gemeinsamen Teilers der Elemente dieser Menge bei steigender Elementanzahl konstant.
- Für die Berechnung des größten gemeinsamen Teilers zweier Elemente entspricht eine Lösung, die den oben angeführten Bedingungen genügt, einer minimalen Lösung bzgl. der L_∞–Norm des Gleichungssystems $a^T \cdot x = \gcd(a)$ mit $a = (a_0, a_1)$ und $x = (x_0, x_1)$.

Dies führt uns zu der folgenden Aufgabenstellung:

MINIMUM GCD MULTIPLIERS

Instanz	Gegeben sei ein Vektor $v = (v_0, \ldots, v_{n-1}) \in \mathbb{Z}^n$ positiver ganzer Zahlen und eine positive ganze Zahl K.
Frage	Existiert ein Vektor $x = (x_0, \ldots, x_{n-1}) \in \mathbb{Z}^n$ mit $\sum_{i=0}^{n-1} x_i \cdot a_i = \gcd(a_0, \ldots, a_{n-1})$ mit $\lvert x_i \rvert < K$ für alle $i \in \{0, \ldots, n-1\}$?

Diese Aufgabe hat sich leider, wie der folgende Satz belegt, als schwer herausgestellt.

10.10. Satz
Das MINIMUM–GCD–MULTIPLIERS–Problem ist NP–vollständig.

Beweis: Siehe [MH94a] ■

Demnach sind wir, wenn wir schon nicht der Lage sind, die kürzeste Darstellung des größten gemeinsamen Teilers bzgl. einer vorgegebenen Metrik zu ermitteln, auf der Suche nach kurzen Darstellungen.

Algorithmus von Bradley
Bereits 1970 hat Bradley in seiner Arbeit [Bra70] den folgenden Algorithmus vorgestellt.

10.11. Algorithmus

mgcd–Algorithmus von Bradley: $mgcd_{Bradley}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$
AUSGABE: $x \in \mathbb{Z}^n$ mit $a_{m-1,*} \cdot x = \gcd(a_{m-1,*})$.

[Vorberechnung]
(1) $v = (a_{m-1,0}, \ldots, a_{m-1,n-1})$;
(2) $g[0] = v_0$;
(3) **for** $(i = 1; i < n; i++)$ **do**
(4) $g[i] = xgcd(g[i-1], v_i, y[i], z[i])$;
(5) **od**
[Balanciertes Rückwärtseinsetzen]
(6) $x_{n-1} = z[n-1]$;
(7) $\hat{y} = y[n-1]$;
(8) **for** $(i = n-2; i > 0; i--)$ **do**
(9) $u = \frac{g[i-1]}{g[i]}; \tilde{u} = \frac{v_i}{g[i]}$;
(10) $w = y[i]; \tilde{w} = z[i]$;
(11) $k = \lfloor \frac{\tilde{w}}{u} \rceil$;
(12) $x_i = \hat{y} \cdot (z[i] - k \cdot u)$;
(13) $\hat{y} = \hat{y} \cdot (w + k \cdot \tilde{u})$;
(14) **od**
(15) $x_0 = \hat{y}$;
(16) **return** (x);

Dieser Algorithmus nutzt geschickt in der Phase des balancierten Rückwärtseinsetzens gewisse Eigenschaften Diophantischer Gleichungen aus, die aus dem folgenden Satz resultieren, um die Koeffizienten der Darstellung zu minimieren.

Ein Nachteil dieser Methode von Bradley liegt darin, daß dieser Algorithmus die Darstellungskoeffizienten auf Kosten des ersten Koeffizienten minimiert, wie die folgenden Abschätzungen belegen [Bra70, Mül94].

Sei $g_i = \gcd(a_0, \ldots, a_i)$, dann gilt:

$$\begin{aligned} |x_0| &\leq \frac{1}{a_0}\left(g_{n-1} + \sum_{i=1}^{n-1} \frac{a_i \cdot g_{i-1}}{2 \cdot g_i}\right) \\ |x_i| &\leq \frac{g_{i-1}}{2 \cdot g_i}, \forall i \in \{1, \ldots, n-1\} \end{aligned}$$

In der Originalversion berechnet der obige Algorithmus keine Transformationsmatrix, sondern lediglich einen Darstellungsvektor. Diesen Nachteil konnten wir aber durch den folgenden Algorithmus beseitigen.

10.12. Algorithmus

mgcd–Algorithmus von Bradley mit Transformationsmatrix: $mgcd_{Bradley}$

EINGABE: $A \in \mathbf{Mat}_{m \times n}(\mathbb{Z})$
AUSGABE: $T \in \mathrm{GL}_n(\mathbb{Z})$ mit $a_{m-1,*} \cdot T = (0, \ldots, 0, \mathrm{ggT}(a_{m-1,*}))$.

[Initialisierung]
(1) $v = (a_{m-1,0}, \ldots, a_{m-1,n-1})$;
(2) $T = I_n$;
[Vorberechnung]
(3) $g_{old} = v_0$;
(4) **for** (i = 1; i < n; i++) **do**
(5) **if** $(a_i \neq 0)$ **then**
(6) $g = xgcd(y, z, g_{old}, v_i)$;
(7) $\mathrm{U} = \begin{pmatrix} I_{i-1} & & & \\ & -\frac{v_i}{g} & y & \\ & \frac{g_{old}}{g} & z & \\ & & & I_{n-i} \end{pmatrix}$;
(8) $T = T \cdot U$;
(9) $g_{old} = g$;
(10) **else**
(11) T.swap_columns(i-1, i);
(12) **fi**
(13) **od**
[Balanciertes Rückwärtseinsetzen]
(14) **for** (i = n - 1, j = n - 2; $i \geq 0 \wedge j \geq 0$; i– –, j– –) **do**
(15) **for** (k = j + 1; k < n; k++) **do**
(16) **if** $(t_{i,j} \neq 0)$ **then**
(17) $div\_rem(q, r, t_{i,k}, t_{i,j})$;
(18) $t_{i,k} = t_{i,k} - q \cdot t_{i,j}$;

```
(19)          if (abs(q)> 0) then
(20)            for (l = 0; l < i; l++) do
(21)              t_{l,k} = t_{l,k} − q · t_{l,j};
(22)            od
(23)          fi
(24)        fi
(25)      od
(26)    od
(27)    return (T);
```

Ergebnisse:
In den Tabellen 10.6, 10.7, 10.8 und 10.9 haben wir wiederum die ermittelten Ergebnisse zusammengestellt, die wir nun im Vergleich zu den Ergebnissen des Modulalgorithmus $mgcd_{linear}$ näher beleuchten.

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	44.6
Bereich (bis)	2.8	1675.9	17788.3	59109.7	6521.61	143.5
abhängig von $L_0(v)$	√	√	√	√	√	√
abhängig von $L_\infty(A)$	–	–	√	√	√	√
abhängig von Eintragsverteilung von A	–	–	–	–	–	–

1. **Anzahl der Nicht–Nulleinträge des Darstellungsvektors x ($L_0(x)$)**
   Bei diesem Modulalgorithmus kann der größte gemeinsame Teiler als Linearkombination von 2 oder 3 Einträgen des Eingabevektors dargestellt werden. Die zufällige Wahl der Einträge und die paarweise ggT–Berechnung sind für diese Beobachtung verantwortlich.

2. **Anzahl der Nicht–Nulleinträge der Transformationsmatrix T ($L_0(T)$)**
   Die Anzahl der Nicht–Nulleinträge der Transformationsmatrix T liegt leicht über der des Modulalgorithmus $mgcd_{linear}$. Dies ist auf den Vorgang des balancierten Rückwärtseinsetzens zurückzuführen.

3. **maximaler Eintrag des Darstellungsvektors x ($L_\infty(x)$)**
   Die Größe des maximalen Eintrags des Darstellungsvektor ist deutlich kleiner im Vergleich zu der entsprechenden Kenngröße des Modulalgorithmus $mgcd_{linear}$. Die Größe des maximalen Eintrags des Darstellungsvektors hängt linear von der Schranke der Eintragsgröße der Startmatrix ab. Die obere Schranke für die Einträge der Startmatrix ist nun ebenfalls eine obere Schranke für die Größe des maximalen Eintrags des Darstellungsvektors. Die Abhängigkeit von der Eintragsdichte wurde ebenfalls abgeschwächt.

4. **maximaler Eintrag der Transformationsmatrix T ($L_\infty(T)$)**
   Die maximalen Einträge der Transformationsmatrix übersteigen auch bei diesem Modulalgorithmus deutlich die jeweiligen maximalen Einträge des Darstellungsvektors.

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2	1457.7	13.9	75	10.0623	141.84
450.1	90.02	99	2	1391.8	12.3	58.9	6.75614	138.072
403.9	80.78	99	2.3	1366.2	12.2	54	6.7818	134.114
350.7	70.14	99	2.2	1203	14.8	53.5	6.26708	129.809
295.6	59.12	98.9	2.6	1150.8	11.7	42.7	4.86609	125.312
253.4	50.68	98.8	2.1	1020.8	10.4	45.6	4.93711	120.249
198.7	39.74	98.7	2	902.1	20.1	69.3	6.36382	114.518
156.3	31.26	98.6	2.3	800.6	16	49.9	3.92168	108.007
97	19.4	98.7	2.3	700.1	12.3	52.3	3.64434	100.968
51.6	10.32	98.1	2.4	606.1	17.4	56.9	2.84128	93.94
29.5	5.9	97.1	2.1	556.7	14.5	60.6	2.35962	91.137
11	2.2	93.6	2.4	521.9	17.1	54.8	1.53899	89.688
5.4	1.08	82.9	2.1	508.8	18.8	67.6	1.40035	89.262
2.8	0.56	62.7	1.8	503.5	10.3	37.4	1.15809	89.097
1.8	0.36	57.3	1.7	501.9	6.4	25.5	1.08946	71.268
0.5	0.1000000	18.8	1	500	1	1	1	44.658

Tabelle 10.6: $mgcd_{Bradley}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.2	1583	177.9	504	54.5843	142.987
449	89.8	997.6	2.4	1481.2	120.6	618.6	74.0198	139.161
398.2	79.64	996	2.4	1390.6	149.1	589.5	60.495	135.032
348.9	69.78	996.6	2.5	1300.2	91.8	511.3	50.8304	130.658
303.2	60.64	996.2	2.4	1178.8	107.6	471.5	47.7056	126.03
243.8	48.76	995.1	2.3	1027.9	118.8	527.8	45.8477	120.851
197.8	39.56	992.8	2.4	941.1	85.2	462.4	34.6843	115.034
149	29.8	992.9	2.3	832.8	128.8	515.4	33.5735	108.491
98	19.6	987.6	2.1	697.3	155.6	574.8	29.5587	101.384
48.4	9.68	978.9	2.6	612.6	52.8	433.6	14.398	94.301
25.7	5.14	962.9	2.2	553.3	143.7	581.9	9.59714	91.41
8.6	1.72	828.9	2.3	516.8	110.4	553.4	5.83243	89.934
6.2	1.24	887.6	2.3	510.9	200.8	600.9	5.6056	89.511
2.6	0.52	624.3	2.1	503.6	67.9	348.4	2.38125	80.379
2	0.4	567.1	1.7	502.3	98.4	391.3	3.01394	80.267
1.1	0.22	257.8	1.2	500.8	49.8	103.9	1.44688	62.475

Tabelle 10.7: $mgcd_{Bradley}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2.2	1594.8	1214.4	5963.4	653.527	143.375
455.2	91.04	9985	2.4	1545	1188.4	4818.7	534.773	139.533
397.3	79.46	9961.8	2.6	1480.1	1320.7	4601.2	442.299	135.233
352.3	70.46	9977.6	2.3	1269.4	1925.7	6039.6	590.938	130.679
301.7	60.34	9968.3	2.6	1212.1	1055.3	4202.9	438.959	126.008
252.5	50.5	9945.4	2.8	1135.6	716.9	4123.7	331.945	120.856
197.8	39.56	9954.8	2.7	990.2	1166.9	4351.4	364.484	114.968
151.5	30.3	9926	2.4	837.9	1721.6	5684.2	426.662	108.462
99.9	19.98	9907.7	2.4	727.7	1134.4	4917.6	276.75	101.42
51.9	10.38	9677.3	2.5	617.6	601.2	3672.5	122.704	94.285
26.4	5.28	9739.2	2.6	559.8	1602.7	5136.3	104.666	91.394
12.8	2.56	9122.1	2.4	526.5	1064	4875.4	56.9176	89.936
4.2	0.84	8115.2	2.2	506.6	1375.9	5327.3	26.771	89.465
3.9	0.78	8317.4	2.4	506.4	870.2	4264.3	22.7909	89.236
1.7	0.34	4758.2	1.4	502	1042.8	2477.8	18.4315	62.49
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	71.343

Tabelle 10.8: $mgcd_{Bradley}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.5	1675.9	8788.3	47214.9	5164.57	143.5
452.1	90.42	99823.7	2.4	1506.9	17788.3	57464.3	6397.69	139.513
399.3	79.86	99763.5	2.7	1494.9	6145.5	40177.3	4583.67	135.359
352.9	70.58	99657.1	2.2	1257.5	15738.6	59109.7	6521.61	132.673
307.4	61.48	99466.2	2.3	1168.4	11912.7	48866.9	4950.56	126.295
251.8	50.36	99431.8	2.5	1073.7	12727.2	45498.9	4204.48	121.158
203.4	40.68	99511.4	2.7	992.6	5659.6	38439	2483.39	115.324
150.7	30.14	99297.6	2.3	835.3	8593.1	54456.1	3557.25	108.745
104.3	20.86	98793.9	2.5	757.3	12143.4	38517.4	2273.73	101.666
54.8	10.96	97747.7	2.5	626.1	11763.6	43873.4	1727.51	94.539
24.6	4.92	96337.4	2.6	554.3	8063.8	47768.2	859.968	91.624
11.5	2.3	92767.5	2.6	524.5	16069.3	54208.5	575.535	90.17
5.1	1.02	79355.4	2.3	509.3	9618.9	38939	193.928	89.726
2.3	0.46	58436	1.9	503.2	6016	23185.9	128.923	71.701
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	62.709
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	62.674

Tabelle 10.9: $mgcd_{Bradley}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

Die Abhängigkeiten zwischen der Größe des maximalen Eintrags des Darstellungsvektors und der Eintragsdichte der Startmatrix bzw. der Größe der Einträge der Startmatrix übertragen sich eins zu eins auf den maximalen Eintrag der Transformationsmatrix.

5. **durchschnittliche Eintragsgröße der Transformationsmatrix T ($\Delta_\infty(T)$)**
   Desweiteren ist die durchschnittliche Eintragsgröße der Nicht–Nulleinträge der Transformationsmatrix deutlich über dem maximalen Eintrag des Darstellungsvektors, bleibt aber auch deutlich hinter dem maximalen Eintrag der Transformationsmatrix zurück. Die oben erwähnten Abhängigkeiten übertragen sich wiederum eins zu eins auf die hier betrachtete Kenngröße.

6. **Laufzeit (t)**
   Die Laufzeiten bewegen sich zwischen 44 bis 134 CPU–Sekunden. Damit ist dieser Modulalgorithmus deutlich langsamer als der Modulalgorithmus $mgcd_{linear}$. Der erhöhte Rechenzeitbedarf ist auf die aufwendige Balancierung der Einträge zurückzuführen. Eine direkte Abhängigkeit zwischen der Laufzeit und der Eintragsdichte ist gut zu erkennen. Der Einfluß der Größe der Einträge der Startmatrix auf die Laufzeit ist minimal.

Algorithmus von Iliopoulos

Aufgrund der Auswertungsreihenfolge des naiven Algorithmus und der damit verbundenen großen Koeffizienten in der Darstellung des gcd scheint es sinnvoll, die Reihenfolge in der Art abzuwandeln, daß ein balancierter binärer Baum entsteht und somit die Tiefe des Auswertungsbaumes reduziert wird. Die Auswertung erfolgt somit gemäß der folgenden Skizze, wobei g_{i-j} für $\gcd(v_i, \ldots, v_j)$ steht.

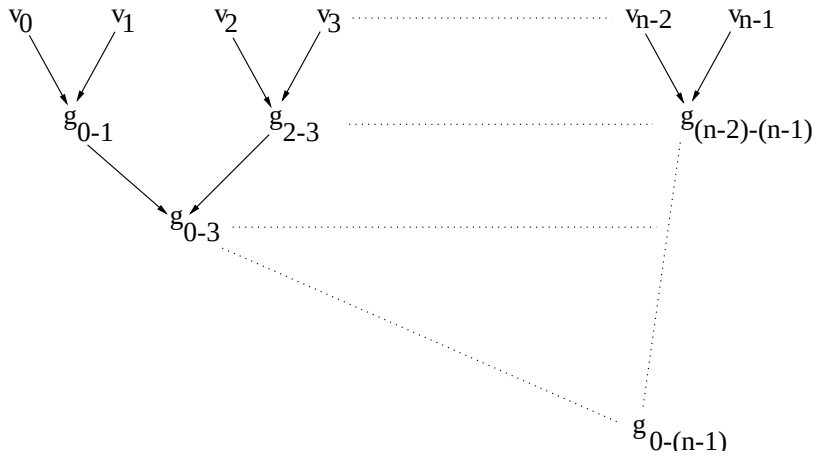


Abbildung 10.2: Verarbeitungsreihenfolge $mgcd_{Ilio}$

Diese Idee macht sich Iliopoulos in seiner Arbeit [Ili89a, Ili89b] zunutze und stellt folgenden Algorithmus vor.

10.13. Algorithmus

mgcd–Algorithmus nach Iliopoulos: $mgcd_{Ilio}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$
AUSGABE: $T \in \mathrm{GL}_n(\mathbb{Z})$ mit $a_{m-1,*} \cdot T = (0, \ldots, 0, \gcd(a_{m-1,*}))$.

```
    [Initialisierung]
(1)   v = (a_{m-1,0}, ..., a_{m-1,n-1});
(2)   T = I_n;
(3)   p = 1; d = 1; s = 2;
    [mgcd–Algorithmus]
(4)   while (p < n) do
(5)     for (i=p; i < n; i += s) do
(6)       j = i - d;
(7)       g = xgcd(v_j, v_i, a, b);
(8)       U = I_n;
(9)       u_{j,j} = a; u_{j,i} = -v_i/g;
(10)      u_{i,j} = b; u_{i,i} = v_j/g;
(11)      T = T · U;
(12)    od
(13)    p = p + s;
(14)    d = d · 2;
(15)    s = s · 2;
(16)  od
(17)  return (T);
```

Dieser Algorithmus führt zu folgenden Schranken für die Größe der Darstellungskoeffizienten in der erweiterten Darstellung des größten gemeinsamen Teilers einer Sequenz ganzer Zahlen.

10.14. Satz
Der Algorithmus $mgcd_{Ilio}$ berechnet g (den größten gemeinsamen Teiler einer Menge ganzer Zahlen $a_0, \ldots, a_{n-1}$, wobei jede dieser Zahlen in der Bitlänge durch β beschränkt ist) und n ganze Zahlen $x_0, \ldots, x_{n-1}$, so daß gilt

$$\sum_{i=0}^{n-1} x_i \cdot a_i = g$$

mit

$$\log_2(\max_i\{x_i\}) = O(\beta \cdot \log_2(n))$$

in $O(n \cdot \log(\beta) \cdot M(\beta))$ elementaren Operationen.

Beweis: Siehe [Ili89a, MH95]. ■

Ergebnisse:
Wie bereits bei den vorangegangenen mgcd–Algorithmen untersuchen wir auch den in diesem Abschnitt vorgestellten Algorithmus in Bezug auf die von uns gewählten Kenngrößen. Die Ergebnisse haben wir in den Tabellen 10.10, 10.11, 10.12 und 10.13 zusammengestellt.

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	0.069
Bereich (bis)	2.5	1544.7	0.3e10	0.7e15	0.1e13	3.217
abhängig von $L_0(v)$	$\surd$	$\surd$	$\surd$	–	–	$\surd$
abhängig von $L_\infty(A)$	–	–	$\surd$	$\surd\surd$	$\surd\surd$	–
abhängig von Eintragsverteilung von A	–	–	–	–	–	–

1. **Anzahl der Nicht–Nulleinträge des Darstellungsvektors x ($L_0(x)$)**
   Der hier betrachtete Modulalgorithmus stellt den größten gemeinsamen Teiles als die Linearkombination von 2 oder 3 Elementen des Eingabevektors dar. Wie bereits erwähnt, ist dies auf die zufällige Wahl der Einträge und auf die zugrundeliegende paarweise ggT–Berechnung zurückzuführen. Im Vergleich zu den vorher betrachtenen Modulalgorithmen zur mgcd–Berechnung entspricht dies dem Verhalten des Modulalgorithmus $mgcd_{linear}$.

2. **Anzahl der Nicht–Nulleinträge der Transformationsmatrix T ($L_0(T)$)**
   Die Anzahl der Nicht–Nulleinträge der Transformationsmatrix T hängt, wie auch bei den beiden anderen bisher betrachteten Modulalgorithmen, an der Anzahl der zur Darstellung des größten gemeinsamen Teilers benötigten Elemente.

3. **maximaler Eintrag des Darstellungsvektors x ($L_\infty(x)$)**
   In Bezug auf die Größe des maximalen Eintrags des Darstellungsvektor verhält sich dieser Modulalgorithmus vergleichbar gut wie der Modulalgorithmus $mgcd_{Bradley}$.

4. **maximaler Eintrag der Transformationsmatrix T ($L_\infty(T)$)**
   Bezüglich des maximalen Eintrags der Transformationsmatrix verhält sich dieser Modulalgorithmus besser als der Modulalgorithmus $mgcd_{linear}$. Er bleibt aber deutlich hinter den Ergebnissen des Modulalgorithmus $mgcd_{Bradley}$ zurück.

5. **durchschnittliche Eintragsgröße der Transformationsmatrix T ($\Delta_\infty(T)$)**
   Ein ähnliches Bild ergibt sich für die durchschnittliche Eintragsgröße der Nicht–Nulleinträge der Transformationsmatrix T. Die durchschnittliche Eintragsgröße der Transformationsmatrizen, die mittels des hier betrachteten Modulalgorithmus berechnet werden, ist deutlich kleiner als die entsprechende Kenngröße des Modulalgorithmus $mgcd_{linear}$. Sie ist aber auch deutlich größer als die entsprechende Kenngröße des Modulalgorithmus $mgcd_{Bradley}$.

6. **Laufzeit (t)**
   Die Laufzeiten dieses Modulalgorithmus bewegen sich zwischen 0,06 bis 3,5 CPU–Sekunden und liegen somit in der Größenordnung des Modulalgorithmus $mgcd_{linear}$.

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2	1492.8	19.3	1295.5	26.7219	3.175
450.1	90.02	99	2.2	1408.3	10.7	3151.5	45.6789	3.138
403.9	80.78	99	2.1	1311.4	13.8	2874.8	59.8096	3.087
350.7	70.14	99	2	1208.6	10.2	7885.7	75.1626	2.977
295.6	59.12	98.9	2.3	1101.8	35	4525.4	74.5874	2.825
253.4	50.68	98.8	2.2	1011.4	51.5	11188.6	78.5162	2.64
198.7	39.74	98.7	2	907.4	15.5	9965	82.1002	2.372
156.3	31.26	98.6	2	816.3	18	7563.1	68.0055	2.106
97	19.4	98.7	2.4	699.1	165.2	5362.1	54.423	1.613
51.6	10.32	98.1	2.3	603.6	13.2	6525	40.6014	1.111
29.5	5.9	97.1	2.5	559.5	39.2	2941	21.6459	0.749
11	2.2	93.6	2.3	521.4	18.3	4668.2	22.5497	0.397
5.4	1.08	82.9	2.1	508.1	12.5	329.2	2.86558	0.238
2.8	0.56	62.7	2	503.8	13.5	334.3	2.33724	0.157
1.8	0.36	57.3	1.6	501.9	3.8	34.6	1.14903	0.118
0.5	0.1000000	18.8	1	500	1	1	1	0.069

Tabelle 10.10: $mgcd_{Ilio}$ Laufzeitergebnisse ($L_\infty(x) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.4	1540.4	126.2	56444.6	318.291	3.195
449	89.8	997.6	2.1	1441	174.9	283746	2944.71	3.152
398.2	79.64	996	2.3	1336.5	1220.8	0.513e7	8584.84	3.082
348.9	69.78	996.6	2.1	1240.8	198.4	0.160e7	6246.49	2.992
303.2	60.64	996.2	2.3	1141.9	169.7	0.105e8	23448.5	2.85
243.8	48.76	995.1	2.1	1013.4	406.1	0.638e7	18543.7	2.625
197.8	39.56	992.8	2.2	917.9	677.1	0.145e8	37186.4	2.374
149	29.8	992.9	2.1	814.9	522.3	0.168e8	41309.3	2.063
98	19.6	987.6	2.5	709.8	4707.6	0.103e8	31683.9	1.63
48.4	9.68	978.9	2.2	601.4	327.4	0.214e7	7750.66	1.05
25.7	5.14	962.9	2.2	554.5	3243.7	0.364e7	12289.7	0.711
8.6	1.72	828.9	2.3	516.2	135.7	136941	574.126	0.331
6.2	1.24	887.6	2.2	511	4743.8	45189.1	276.611	0.26
2.6	0.52	624.3	2.1	503.7	101.4	38368.3	120.617	0.154
2	0.4	567.1	1.7	502.3	2276.8	31762.8	154.116	0.128
1.1	0.22	257.8	1.2	500.8	49.8	745.7	3.45387	0.094

Tabelle 10.11: $mgcd_{Ilio}$ Laufzeitergebnisse ($L_\infty(x) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2	1532.7	2032	0.139e7	4146.17	3.194
455.2	91.04	9985	2	1447.5	1513	0.275e8	214181	3.166
397.3	79.46	9961.8	2.2	1338.5	147643	0.346e9	896569	3.094
352.3	70.46	9977.6	2.1	1241.2	16123.1	0.169e10	0.272e7	2.992
301.7	60.34	9968.3	2.2	1137.5	95987.9	0.484e10	0.706e7	2.855
252.5	50.5	9945.4	2.1	1040.5	21939.8	0.295e10	0.581e7	2.661
197.8	39.56	9954.8	2.4	924.4	40248.1	0.696e10	0.196e8	2.402
151.5	30.3	9926	2.1	822.9	194267	0.413e12	0.652e9	2.057
99.9	19.98	9907.7	2.5	713.3	0.202e8	0.168e12	0.343e9	1.665
51.9	10.38	9677.3	2.4	610	478412	0.981e10	0.215e8	1.096
26.4	5.28	9739.2	2.3	554.3	30018	0.113e9	391297	0.698
12.8	2.56	9122.1	2.3	525.1	111023	0.521e9	0.125e7	0.417
4.2	0.84	8115.2	2	506.4	1515.6	0.237e7	5786.7	0.211
3.9	0.78	8317.4	2.2	506.3	127204	0.899e7	34942.1	0.191
1.7	0.34	4758.2	1.4	502	1064.8	0.186e7	14240.6	0.116
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.094

Tabelle 10.12: $mgcd_{Ilio}$ Laufzeitergebnisse ($L_\infty(x) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.2	1544.7	20071.1	0.688e7	32927	3.217
452.1	90.42	99823.7	2.4	1449.5	0.159e8	0.433e12	0.562e9	3.174
399.3	79.86	99763.5	2.1	1352.6	258505	0.141e13	0.176e10	3.098
352.9	70.58	99657.1	2.3	1253.1	0.749e7	0.739e12	0.148e10	3.041
307.4	61.48	99466.2	2.4	1159.9	14769.9	0.650e13	0.102e11	2.865
251.8	50.36	99431.8	2	1037.9	20823.1	0.391e13	0.937e10	2.668
203.4	40.68	99511.4	2.1	931.5	170626	0.695e15	0.105e13	2.414
150.7	30.14	99297.6	2.3	826	0.148e8	0.881e13	0.278e11	2.087
104.3	20.86	98793.9	2.3	722.8	0.118e8	0.546e13	0.113e11	1.697
54.8	10.96	97747.7	2.1	615.3	0.222e7	0.444e13	0.125e11	1.139
24.6	4.92	96337.4	2.5	552.6	0.328e10	0.734e14	0.236e12	0.649
11.5	2.3	92767.5	2	522.2	17939.8	0.293e12	0.804e9	0.405
5.1	1.02	79355.4	2	508.6	16343.7	0.816e9	0.304e7	0.239
2.3	0.46	58436	1.7	503	5607	0.260e9	788225	0.136
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.084
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.088

Tabelle 10.13: $mgcd_{Ilio}$ Laufzeitergebnisse ($L_\infty(x) < 100000$)

gcd–Tree Algorithmus von Majewski und Havas
Kombiniert man die Idee der Verwendung eines binären Baumes mit der Idee des intelligenten Rückwärtseinsetzens, so führt dies zu dem folgenden Algorithmus von Majewski und Havas [MH94a].

10.15. Algorithmus

mgcd–Tree Algorithmus von Majewski und Havas:
$mgcd_{Tree}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbf{Z})$
AUSGABE: $x \in \mathbf{Z}^n$ mit $a_{m-1,*} \cdot x = \gcd(a_{m-1,*})$.

[Initialisierung]
(1) **for** (i = 0; i < n; i++) **do**
(2) $g_{n+i} = a_{m-1,i}$;
(3) **od**
[Bottom-Up Phase]
(4) **for** (i = n-1; i > 0; i– –) **do**
(5) $g_i = xgcd(g_{2\cdot i}, g_{2\cdot i+1}, y_{2\cdot i}, y_{2\cdot i+1})$;
(6) **od**
[Top-Down Phase]
(7) $z_2 = y_2$;
(8) $z_3 = y_3$;
(9) **for** (i = 2; i < n; i++) **do**
(10) $g_i = g_i \cdot z_i$;
(11) $u = \frac{g_{2\cdot i}}{g_i}; \tilde{u} = \frac{g_{2\cdot i+1}}{g_i}$;
(12) $w = y_{2\cdot i} \cdot z_i; \tilde{w} = y_{2\cdot i+1} \cdot z_i$;
(13) $k = \lfloor \frac{\tilde{w}}{u} \rceil$;
(14) $z_{2\cdot i+1} = \tilde{w} - k \cdot u$;
(15) $z_{2\cdot i} = w + k \cdot \tilde{u}$;
(16) **od**
(17) **return** (x);

Zur Berechnung des mgcd von n ganzen Zahlen erstellt der obige Algorithmus einen binären Baum ("gcd - tree") mit $(2 \cdot n - 1)$ Knoten. Diese Knoten werden von oben nach unten und von links nach rechts beginnend bei 1 durchnumeriert, so daß die Kinder eines Knotens i die Nummer $2 \cdot i$ und $2 \cdot i + 1$ tragen. Im weiteren Verlauf der Beschreibung verwenden wir die Nummer eines Knotens als Index, um zu kennzeichnen, welche Werte an welcher Stelle des binären Baumes gespeichert werden.

Den Knoten n bis $2 \cdot n - 1$ (Blätter des binären Baums) werden zunächst die Werte des Eingabevektors $v = (v_0, \ldots, v_{n-1})$ zugeordnet, d.h. $g_n = v_0, \ldots, g_{2\cdot n-1} = v_{n-1}$. Am Ende des Algorithmus enthält die Wurzel des binären Baumes $g_1 = \gcd(v)$ und die Blätter die Koeffizienten $x_0, \ldots, x_{n-1}$ der Darstellung.

Der Algorithmus arbeitet in zwei Phasen.

1. **Bottom–Up**
   Die Bottom–Up–Phase startet bei den Blättern des binären Baumes und arbeitet sich stufenweise bis zur Wurzel des Baumes vor. In jeder Stufe berechnet der Algorithmus den größten gemeinsamen Teiler inklusive der Darstellung zweier Elemente mit gleichem Vaterknoten und speichert die Darstellungskoeffizienten an den jeweiligen Knoten ab. Für zwei Knoten $4 \cdot i$ und $4 \cdot i + 1$ bedeutet dies, der Algorithmus berechnet $g_{2 \cdot i} = g_{4 \cdot i} \cdot y_{4 \cdot i} + g_{4 \cdot i+1} \cdot y_{4 \cdot i+1}$ und speichert $g_{2 \cdot i}$, $y_{4 \cdot i}$ und $y_{4 \cdot i+1}$ an den Knoten $2 \cdot i$, $4 \cdot i$ und $4 \cdot i + 1$. Diese Phase endet sobald die Wurzel des Baumes erreicht ist.

   10.16. Bemerkung
   Diese Berechnungen können parallel durchgeführt werden.

2. **Top–Down**
   Die Top–Down–Phase beginnt bei Knoten 4 (unter der Annahme, daß $n \geq 3$) und arbeitet sich stufenweise zu den Blättern des Baumes vor. Wiederum betrachten wir zu einem Zeitpunkt lediglich zwei Knoten mit gleichem Vaterknoten. Für zwei Knoten $4 \cdot i$ und $4 \cdot i + 1$ berechnet der Algorithmus zwei Koeffizienten $z_{4 \cdot i}$ und $z_{4 \cdot i+1}$, so daß gilt:

   $$z_{4 \cdot i} \cdot g_{4 \cdot i} + z_{4 \cdot i+1} \cdot g_{4 \cdot i+1} = g_{2 \cdot i} \cdot z_{2 \cdot i}$$

   Der Wert $z_{2 \cdot i}$ entspricht entweder direkt dem Wert $y_{2 \cdot i}$ oder dem Wert der bereits in einem vorangegangenen Schritt der Top–Down–Phase berechnet wurde. Zur Berechnung von $z_{4 \cdot i}$ und $z_{4 \cdot i+1}$ macht sich dieser Algorithmus die Tatsache $y_{4 \cdot i} \cdot g_{4 \cdot i} + y_{4 \cdot i+1} \cdot g_{4 \cdot i+1} = g_{2 \cdot i}$ zunutze. Der Algorithmus multipliziert jede Seite der vorangegangenen Gleichung mit $z_{2 \cdot i}$ und verwendet die im Algorithmus von-Bradley verwendete Technik zur Berechnung von *kleinen* Koeffizienten. Diese Phase endet, sobald alle Werte der Blätter berechnet wurden.

Leider ist es uns bisher nicht gelungen, einen Algorithmus anzugeben, der parallel eine unimodulare Transformationsmatrix berechnet. Aus diesem Grund können wir diesem Algorithmus nicht direkt als Modulalgorithmus verwenden.

Eine Bewertung dieses Algorithmus in Bezug auf die für mgcd–Algorithmen verwendeten Kenngrößen entfällt somit.

Minimierung der Anzahl der Darstellungskoeffizienten
In den bisher vorgestellten Algorithmen zur Berechnung des größten gemeinsamen Teilers einer Menge von Zahlen haben wir versucht, eine minimale Lösung bzgl. der L_∞–Norm zu finden. In diesem Abschnitt widmen wir uns der Aufgabe, eine minimale Darstellung bzgl. der L_0–Metrik zu berechnen. Ein optimaler Lösungsvektor x bzgl. dieser Metrik enthält möglichst viele Nulleinträge. Desweitern muß für eine optimale Lösung gelten:

$$L_0(x) \leq \log_2(\min(\{a_{m-1,i} | 0 \leq i < n, a_{m-1,i} > 0\}))$$

Es ist offensichtlich, daß im allgemeinen eine minimale Lösung bzgl. der L_0–Metrik nicht einer optimalen Lösung bzgl. der L_∞–Norm entspricht.

Formal ausgedrückt widmen wir uns in diesem Abschnitt der folgenden Aufgabe:

MINIMUM GCD SET

Instanz	Gegeben sei eine Menge ganzer Zahlen $M = \{v_0, \ldots, v_{n-1}\}$.
Frage	Enthält M eine echte Teilmenge M' mit $\#M' < n$ und $\gcd(x \in M') = \gcd(x \in M)$?

Auch diese Aufgabe hat sich als schwer herausgestellt, wie der folgende Satz belegt.

10.17. Satz
Das MINIMUM–GCD–SET–Problem ist NP–vollständig.

Beweis: Siehe [MH94a] ■

10.18. Bemerkung
Die NP–Vollständigkeit des MINIMUM–GCD–SET–Problems widerspricht auf den ersten Blick der praktischen Erfahrung, daß der größte gemeinsame Teiler einer Menge von zufälligen Zahlen dem größten gemeinsamen Teiler von zwei oder drei dieser Zahlen entspricht. Desweiteren besteht gemäß eines Resultats von E. Césaro die realistische Chance, daß zwei der Elemente bereits den größten gemeinsamen Teiler 1 besitzen. Daher erwarten wir auf der einen Seite, durch einfache Inspektion des größten gemeinsamen Teilers von einigen Zahlenpaaren eine kurze Lösung zu finden. Aber auf der anderen Seite zeigt der Beweis der NP–Vollständigkeit, wenn nicht P = NP gilt, daß es manchmal exponentiell lange dauern kann, eine kurze Lösung zu finden.

Mit der Zielsetzung einen Algorithmus zu entwickeln, der die Anzahl der Elemente in der Darstellung des *gcd* minimiert, ergänzen wir den Algorithmus $mgcd_{linear}$ um die Heuristik, daß man zuerst den größten gemeinsamen Teiler eines festen Elementes mit verschiedenen anderen Elementen berechnet und dann als Startpunkt dasjenige wählt, welches den minimalen *gcd* ergab.

10.19. Algorithmus

Linear iterativer mgcd–Algorithmus: $mgcd_{opt}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$
AUSGABE: $T \in \mathrm{GL}_n(\mathbb{Z})$ mit $a_{m-1,*} \cdot T = (0, \ldots, 0, \gcd(a_{m-1,*}))$.

[Initialisierung]
(1) $v = (a_{m-1,0}, \ldots, a_{m-1,n-1})$;
(2) $T = I_n$;

(3) **for** (i = 0; $v_i = 0 \wedge$ i < n; i++) **do**
(4) **od**
(5) **if** ($i \neq n-1$) **then**
(6) swap(v_i, v_0);
(7) T.swap_columns(i, 0);
(8) **fi**
[Sortierung]
(9) $j = 1$;
(10) $g_{min} = gcd(v_0, v_j)$;
(11) **for** (i = 2; i < n; i++) **do**
(12) $g_{tmp} = gcd(v_0, v_i)$;
(13) **if** ($g_{tmp} < g_{min}$) **then**
(14) $j = i$;
(15) **fi**
(16) **od**
(17) swap(v_j, v_1);
(18) T.swap_columns(j,1);
[mgcd–Algorithmus]
(19) **for** (i= 1; i < n; i++) **do**
(20) g = xgcd(v_{i-1},v_i, a, b);
(21) U = $\begin{pmatrix} I_{i-1} & & & \\ & -\frac{v_i}{g} & a & \\ & \frac{v_{i-1}}{g} & b & \\ & & & I_{n-i} \end{pmatrix}$;
(22) $v = v \cdot U$;
(23) $T = T \cdot U$;
(24) **od**
(25) **return** (T);

Ergebnisse:
Dieser Algorithmus liefert uns die in den Tabellen 10.14, 10.15, 10.16 und 10.17 zusammengefaßten Ergebnisse.

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	0.046
Bereich (bis)	2.9	1795.6	0.9e13	0.9e18	0.18e18	3.167
abhängig von $L_0(v)$	√	√	√	√	√	√
abhängig von $L_\infty(A)$	–	–	√	√√	√√	√
abhängig von Eintragsverteilung von A	–	–	–	–	–	–

1. **Anzahl der Nicht–Nulleinträge des Darstellungsvektors** x **(**$L_0(x)$**)**
Der größte, gemeinsame Teiler wird als die Linearkombination von 1, 2 oder 3 Elementen des Eingabevektors dargestellt. Damit entspricht das Verhalten dem der vorangehenden Modulalgorithmen.

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2	1488.6	14.2	1405.8	359.949	3.075
450.1	90.02	99	2	1387.3	12.6	1247.4	287.579	2.772
403.9	80.78	99	2.1	1336.1	19.7	1950.3	424.636	2.489
350.7	70.14	99	2	1192.6	19.2	1900.8	418.297	2.172
295.6	59.12	98.9	2.1	1113.3	74.5	7320.4	1374.86	1.829
253.4	50.68	98.8	2.1	1021.5	38.9	3847.4	582.14	1.557
198.7	39.74	98.7	2.1	911.7	44.5	4393.4	807.246	1.23
156.3	31.26	98.6	2.1	824.4	29	2863.3	371.21	0.966
97	19.4	98.7	2.5	737	103.2	10210.3	1000.6	0.612
51.6	10.32	98.1	2.4	618.8	92.4	8940.5	469.615	0.333
29.5	5.9	97.1	2.1	559	35.5	3419	136.042	0.208
11	2.2	93.6	2.3	522.3	32.2	3038.2	36.0094	0.103
5.4	1.08	82.9	2.2	508.4	17.5	1068	7.88906	0.075
2.8	0.56	62.7	2	503.9	87.1	929	7.47668	0.058
1.8	0.36	57.3	1.7	501.9	27.7	70.4	1.25503	0.049
0.5	0.1000000	18.8	1	500	1	1	1	0.046

Tabelle 10.14: $mgcd_{opt}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2	1497	146.8	146527	35931.1	3.116
449	89.8	997.6	2	1395.4	155.6	155230	36074.8	2.797
398.2	79.64	996	2.4	1452.8	11975.4	0.119e8	0.260e7	2.494
348.9	69.78	996.6	2	1194.6	82.8	82620.5	17870.3	2.179
303.2	60.64	996.2	2.3	1196.9	13594.1	0.136e8	0.302e7	1.9
243.8	48.76	995.1	2.2	1032.9	1613.1	0.161e7	310612	1.524
197.8	39.56	992.8	2.2	932.1	692	684852	106570	1.233
149	29.8	992.9	2.1	813.1	679.4	676940	103141	0.93
98	19.6	987.6	2.3	723	1732.8	0.171111e7	176245	0.618
48.4	9.68	978.9	2.4	613.2	2759.5	0.269e7	132245	0.321
25.7	5.14	962.9	2.3	555.1	3277.6	0.324e7	113300	0.189
8.6	1.72	828.9	2.5	517.9	14325.1	0.103e8	85154.7	0.09
6.2	1.24	887.6	2.4	511.8	4503.1	0.244e7	12764.7	0.077
2.6	0.52	624.3	2.1	503.7	103.7	36733.6	118.865	0.057
2	0.4	567.1	1.7	502.4	651.9	554056	1385.2	0.053
1.1	0.22	257.8	1.3	501	29.9	24087.3	86.2878	0.049

Tabelle 10.15: $mgcd_{opt}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2.1	1547.6	5495	0.549e8	0.114e8	3.141
455.2	91.04	9985	2.3	1544.2	0.636e8	0.636e12	0.149e12	2.87
397.3	79.46	9961.8	2.1	1331.7	175281	0.175e10	0.468e9	2.512
352.3	70.46	9977.6	2.2	1269.5	21635.6	0.216e9	0.347e8	2.222
301.7	60.34	9968.3	2.2	1161.3	556119	0.555e10	0.811e9	1.902
252.5	50.5	9945.4	2.2	1051.8	925500	0.924e10	0.174e10	1.591
197.8	39.56	9954.8	2.7	1029.3	0.325e11	0.324e15	0.589e14	1.255
151.5	30.3	9926	2.4	861.4	152306	0.151949e10	0.166931e9	0.955
99.9	19.98	9907.7	2.7	762.9	0.115e11	0.114e15	0.139e14	0.638
51.9	10.38	9677.3	2.8	640	0.141e9	0.127e13	0.616e11	0.342
26.4	5.28	9739.2	2.2	555.4	175006	0.170e10	0.549e8	0.193
12.8	2.56	9122.1	2.3	526.3	143880	0.131e10	0.129e8	0.115
4.2	0.84	8115.2	2.5	507.5	0.402e7	0.264e11	0.872e8	0.062
3.9	0.78	8317.4	2.2	506.6	496892	0.336e10	0.258e8	0.063
1.7	0.34	4758.2	1.5	502.2	409162	0.141e10	0.405e7	0.053
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.049

Tabelle 10.16: $mgcd_{opt}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.6	1795.6	0.935e13	0.935e18	0.180e18	3.167
452.1	90.42	99823.7	2.1	1447.6	0.582e8	0.581e13	0.125e13	2.848
399.3	79.86	99763.5	2.4	1455.2	0.933e7	0.931e12	0.166e12	2.542
352.9	70.58	99657.1	2.2	1271.7	0.803e7	0.793e12	0.151e12	2.249
307.4	61.48	99466.2	2.3	1204.1	0.206e8	0.204e13	0.323e12	1.945
251.8	50.36	99431.8	2.2	1051.1	0.488e8	0.484e13	0.996e12	1.595
203.4	40.68	99511.4	2.1	924.9	955049	0.949e11	0.157e11	1.283
150.7	30.14	99297.6	2.3	843.2	0.112e8	0.110e13	0.129e12	0.951
104.3	20.86	98793.9	2.6	763.4	0.203e9	0.196e14	0.152e13	0.664
54.8	10.96	97747.7	2.3	623.7	0.532e8	0.521e13	0.280e12	0.358
24.6	4.92	96337.4	2.9	569.4	0.730e12	0.721e17	0.187e16	0.185
11.5	2.3	92767.5	2	521	17939.8	0.5	0.269e8	0.106
5.1	1.02	79355.4	2.1	508.4	0.591e7	0.286e11	0.106e9	0.071
2.3	0.46	58436	1.7	503	5079.5	0.213e9	0.107e7	0.055
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.049
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.047

Tabelle 10.17: $mgcd_{opt}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

2. **Anzahl der Nicht–Nulleinträge der Transformationsmatrix T ($L_0(T)$)**
   Die Anzahl der Nicht–Nulleinträge der Transformationsmatrix T liegt in der Größenordnung der entsprechenden Kenngröße der Modulalgorithmen $mgcd_{linear}$ und $mgcd_{Ilio}$.

3. **maximaler Eintrag des Darstellungsvektors x ($L_\infty(x)$)**
   In Bezug auf die Größe des maximalen Eintrags des Darstellungsvektor verhält sich dieser Modulalgorithmus vergleichbar dem Modulalgorithmus $mgcd_{linear}$.

4. **maximaler Eintrag der Transformationsmatrix T ($L_\infty(T)$)**
   Bezüglich des maximalen Eintrags der Transformationsmatrix verhält sich dieser Modulalgorithmus wie der Modulalgorithmus $mgcd_{linear}$. Dies beinhaltet sowohl die Eintragsgröße als auch die Abhängigkeit zur Eintragsdichte und zur maximalen Eintragsgröße der Eingabematrix.

5. **durchschnittliche Eintragsgröße der Transformationsmatrix T ($\Delta_\infty(T)$)**
   Ein ähnliches Bild ergibt sich für die durchschnittliche Eintragsgröße der Nicht–Nulleinträge der Transformationsmatrix T. Das Verhalten entspricht im wesentlichen dem des Modulalgorithmus $mgcd_{linear}$.

6. **Laufzeit (t)**
   Die Laufzeiten dieses Modulalgorithmus bewegen sich zwischen 0,046 bis 3,167 CPU–Sekunden und liegen somit in der Größenordnung des Modulalgorithmus $mgcd_{linear}$. Es ist lediglich ein leichter Laufzeitgewinn feststellbar. Dies ist auf die zufällige Wahl der Einträge der Startmatrix zurückzuführen.

10.20. Bemerkung
Diese heuristische Erweiterung ist exemplarisch zu verstehen. Prinzipiell kann man jeden der bisher vorgestellten Algorithmen und jeden der noch folgenden mgcd–Algorithmen um diese heuristische Komponente erweitern.

10.1.1.2 mgcd–Algorithmen auf der Basis des Algorithmus von Blankinship

Naiver Algorithmus
Ausgangspunkt unserer weiteren Betrachtungen ist die folgende Variante des Algorithmus von Blankinship [Bla63, HM94a].

10.21. Algorithmus

mgcd Algorithmus von Blankinship: $mgcd_{Blankin}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$
AUSGABE: $T \in \mathrm{GL}_n(\mathbb{Z})$ mit $a_{m-1,*} \cdot T = (0, \ldots, 0, \gcd(a_{m-1,*}))$.

[Initialisierung]
(1) $T = I_n$

```
(2)   v = (a_{m-1,0}, ..., a_{m-1,n-1});
      [mgcd–Algorithmus]
(3)   j = index_abs_min(v);                  //|v_j| = min(|v_0|, ..., |v_{n-1}|)
(4)   swap(v_j, v_{n-1});                    //v_j ⟷ v_{n-1}
(5)   T.swap_columns(n − 1, j);
(6)   while (∃l : v_l ≠ 0 ∧ l ≠ n − 1) do
(7)     pos_div_rem(q, r, v_l, v_{n-1});     //v_l = q · v_{n-1} + r ∧ r ≥ 0
(8)     v_l = v_l − q · v_{n-1};             //v_l = r
(9)     t_{*,l} = t_{*,l} − q · t_{*,n-1};
(10)    swap(v_l, v_{n-1});                  //v_l ⟷ v_{n-1}
(11)    T.swap_columns(n − 1, l);
(12)  od
(13)  return (T);
```

10.22. Bemerkung
Die Funktion *index_abs_min* liefert den Index des betragsmäßig kleinsten Elementes eines Vektors ganzer Zahlen, der dieser Funktion als Parameter übergeben wird. Die Funktion *swap(a,b)* vertauscht die Elemente a und b.

Dieser Algorithmus ermittelt zunächst das betragsmäßig kleinste Nicht–Nullelement der letzten Zeile der Matrix A, welche wir im folgenden als Arbeits- bzw. Pivotzeile bezeichnen. Das gefundene Element bezeichnen wir im folgenden als Pivotelement. Anschließend wählt er ein weiteres Element der Arbeitszeile ungleich Null aus und reduziert dieses mittels einer geeigneten Linearkombination modulo dem Pivotelement. Ist der so berechnete Eintrag ungleich Null wird er zum neuen Pivotelement. Diese Vorgehensweise wird solange wiederholt, bis lediglich ein Element ungleich dem Nullelement in der Arbeitszeile übrig bleibt. Dieses Element entspricht dann dem größten gemeinsamen Teiler der Elemente der Arbeitszeile.

Ergebnisse:
Der obige Algorithmus führt zu den in den Tabellen 10.18, 10.19, 10.20 und 10.21 zusammengefaßten Ergebnissen.

Man stellt fest, daß er sich in allen betrachten Kenngrößen wie der Modulalgorithmus $mgcd_{linear}$ verhält. Dies ist leicht erklärbar. Wenn man sich den obigen Algorithmus genau betrachtet, berechnet er zunächst den größten gemeinsamen Teiler des betragsmäßig kleinsten und eines beliebigen Elementes der letzten Zeile der Startmatrix. Dieses beliebige Element entspricht dem ersten gefundenen Nicht–Nullelement. Anschließend berechnet er den ggT des im vorigen Schritt erhaltenen Ergebnisses, welches aufgrund der Struktur des Algorithmus dem kleinsten Element der Arbeitszeile entspricht, und eines weiteren beliebigen Elementes der Arbeitszeile. Damit entspricht diese Vorgehensweise (in etwa) der Vorgehensweises des Modulalgorithmus $mgcd_{linear}$. Die zufällige Wahl der Einträge ist nun dafür verantwortlich, daß wir, trotz der geringen Unterschiede, annähernd gleiche Kennwerte erhalten.

Auf eine detaillierte Beschreibung der Kennwerte verzichten wir an dieser Stelle und verweisen auf die Betrachtungen zu den Ergebnissen des Modulalgorithmus $mgcd_{linear}$.

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2.1	1537.5	34.5	3415.5	736.23	3.078
450.1	90.02	99	2.2	1476.4	17.6	1742.4	410.915	2.783
403.9	80.78	99	2.4	1457.1	47.9	4742.1	994.343	2.501
350.7	70.14	99	2.2	1260.7	31.1	3078.9	716.1	2.176
295.6	59.12	98.9	2.3	1171.2	71.3	7003.6	1263.89	1.841
253.4	50.68	98.8	2.4	1095.6	93.4	9227.9	1271.95	1.579
198.7	39.74	98.7	2.3	951.2	73.1	7203.1	1106.24	1.233
156.3	31.26	98.6	2.2	839.3	45.6	4508.2	568.87	0.973
97	19.4	98.7	2.5	737	105.4	10423.1	1015.43	0.612
51.6	10.32	98.1	2.3	614.8	74.4	7204.9	537.169	0.335
29.5	5.9	97.1	2.2	560.9	40.4	3869.1	150.655	0.205
11	2.2	93.6	2.3	522.3	32.2	3038.2	36.0094	0.103
5.4	1.08	82.9	2.2	508.4	17.5	1068	7.88906	0.068
2.8	0.56	62.7	2	503.9	87.1	929	7.47668	0.056
1.8	0.36	57.3	1.7	501.9	27.7	70.4	1.25503	0.051
0.5	0.1000000	18.8	1	500	1	1	1	0.044

Tabelle 10.18: $mgcd_{Blankinship}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.5	1745.4	411355	0.411e9	0.918e8	3.13
449	89.8	997.6	2.4	1573.7	3892.5	0.389e7	850292	2.814
398.2	79.64	996	2.6	1533.1	4318.4	0.431e7	673	2.504
348.9	69.78	996.6	2.3	1299.7	1585.9	0.158e7	297680	2.192
303.2	60.64	996.2	2.4	1224.6	11312.8	0.113e8	0.179e7	1.911
243.8	48.76	995.1	2.4	1079.2	1958.2	0.196e7	346711	1.532
197.8	39.56	992.8	2.3	952.3	380.2	375937	65913.4	1.245
149	29.8	992.9	2.5	871.3	207480	0.206e9	0.257e8	0.938
98	19.6	987.6	2.4	731.9	3695	0.359e7	364767	0.621
48.4	9.68	978.9	2.3	608.5	2456.1	0.239e7	114215	0.319
25.7	5.14	962.9	2.4	557.9	3486.9	0.345e7	118587	0.186
8.6	1.72	828.9	2.5	517.9	14330.7	0.103e8	85236.8	0.091
6.2	1.24	887.6	2.4	511.8	4503.1	0.244e7	12764.7	0.073
2.6	0.52	624.3	2.1	503.7	103.7	36733.6	118.865	0.057
2	0.4	567.1	1.7	502.4	651.9	554056	1385.2	0.053
1.1	0.22	257.8	1.3	501	29.9	24087.3	86.2878	0.05

Tabelle 10.19: $mgcd_{Blankinship}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2.7	1846	0.969e7	0.969e11	0.157e11	3.177
455.2	91.04	9985	2.7	1725.3	0.652e8	0.652e12	0.136e12	2.905
397.3	79.46	9961.8	2.4	1448.4	0.125e7	0.125e11	0.252e10	2.535
352.3	70.46	9977.6	2.2	1273.7	0.759e7	0.756e11	0.153e11	2.24
301.7	60.34	9968.3	2.4	1221	604822	0.604e10	0.856e9	1.919
252.5	50.5	9945.4	2.6	1157.6	0.274e8	0.273e12	0.332e11	1.606
197.8	39.56	9954.8	2.8	1048.3	0.171e9	0.170e13	0.211e12	1.262
151.5	30.3	9926	2.4	861.7	109295	0.109e10	0.119e9	0.961
99.9	19.98	9907.7	2.9	782.8	0.115e11	0.114e15	0.136e14	0.641
51.9	10.38	9677.3	2.7	635.4	0.141e9	0.127e13	0.620e11	0.342
26.4	5.28	9739.2	2.2	555.4	175006	0.170e10	0.549e8	0.19
12.8	2.56	9122.1	2.3	526.3	143880	0.131e10	0.129e8	0.115
4.2	0.84	8115.2	2.5	507.5	0.402e7	0.264e11	0.871e8	0.064
3.9	0.78	8317.4	2.2	506.6	496892	0.335e10	0.258e8	0.064
1.7	0.34	4758.2	1.5	502.2	409162	0.141e10	0.405e7	0.052
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.047

Tabelle 10.20: $mgcd_{Blankinship}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.9	1945.2	0.696e16	0.696e21	0.153e21	3.191
452.1	90.42	99823.7	2.5	1626	0.139e9	0.139e14	0.264e13	2.885
399.3	79.86	99763.5	2.5	1494.8	0.147e9	0.146e14	0.310e13	2.549
352.9	70.58	99657.1	2.4	1338.9	0.366e8	0.365e13	0.737e12	2.272
307.4	61.48	99466.2	2.5	1260.6	0.838e11	0.831e16	0.156e16	1.967
251.8	50.36	99431.8	2.2	1051.2	0.477e8	0.473e13	0.969e12	1.599
203.4	40.68	99511.4	2.1	926.1	0.132e7	0.132e12	0.196e11	1.287
150.7	30.14	99297.6	2.6	887.6	0.299e8	0.296e13	0.325e12	0.964
104.3	20.86	98793.9	2.6	763.4	0.203e9	0.196e14	0.152e13	0.668
54.8	10.96	97747.7	2.4	628	0.681e8	0.667e13	0.359e12	0.357
24.6	4.92	96337.4	2.8	568	0.729e12	0.721e17	0.188e16	0.184
11.5	2.3	92767.5	2	521	17939.8	0.165e10	0.269e8	0.104
5.1	1.02	79355.4	2.1	508.4	0.591e7	0.286e11	0.106e9	0.069
2.3	0.46	58436	1.7	503	5079.5	0.213e9	0.107e7	0.054
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.046
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.049

Tabelle 10.21: $mgcd_{Blankinship}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

Minimierung der Größe der Einträge der Eingabematrix
Die zu Beginn dieses Kapitels vorgestellten Kernalgorithmen illustrieren, daß zur Berechnung der HNF in jeder Zeile eine Berechnung des gcd und einer entsprechenden Transformationsmatrix nötig ist. Demzufolge propagieren sich Veränderungen der Einträge der Matrix A von einer mgcd–Berechnung in die folgende.

Es scheint somit sinnvoll, anstatt das Augenmerk auf die Berechnung einer möglichst minimalen Darstellung des größten gemeinsamen Teilers einer Menge von ganzen Zahlen zu legen, zu versuchen, eine *geeignete* Darstellung des größten gemeinsamen Teilers zu finden.

Anders ausgedrückt, bedeutet dies:

Suche einen ganzzahligen Vektor $x = (x_0, \ldots, x_{n-1})$ mit

$$\begin{array}{rcl} (a_{m-1,0}, \ldots, a_{m-1,n-1}) \cdot x & = & \gcd(a_{m-1,0}, \ldots, a_{m-1,n-1}) \\ & \text{mit} & \\ \max_i |a_{i,*} \cdot x| & & \textit{minimal}! \end{array}$$

P. van Emde Boas hat in seiner Arbeit [vEB81] bewiesen, daß das Problem, den maximalen Eintrag eines Vektors v zu minimieren, wobei v als Linearkombination einer Menge von n Vektoren dargestellt ist, NP-vollständig ist. Dies bedeutet für die hier vorgestellte Optimierungsaufgabe, daß wir wiederum auf ein schweres Problem gestoßen sind und gezwungen sind, mit Approximationen an die optimale Lösung zu arbeiten.

George Havas und Bohdan S. Majewski gehen in diversen Publikationen [HM94b, HM94a] noch einen Schritt weiter. Anstatt lediglich eine *geeignete* Darstellung des größten gemeinsamen Teiles zu suchen, versuchen sie, eine *geeignete* Transformationsmatrix als Ganzes zu finden.

Ausgangspunkt ihrer Algorithmen ist die im vorangegangenen Abschnitt dargestellte Variante des Algorithmus von Blankinship [Bla63, HM94a].

Durch einfache Modifikationen wird der Algorithmus von Blankinship so verändert, daß er in Bezug auf die Größe der Zwischeneinträge und auf die Anzahl der neu entstehenden Einträge ein besseres Verhalten an den Tag legt. Im folgenden stellen wir die Modifikationen im einzelnen im Kontext des verfolgten Ziels vor.

best–remainder Strategie
In der Arbeit [HHR93] schlagen Havas, Holt und Rees die folgenden Modifikationen vor, die sie als *best–remainder*–Strategie bezeichnen:

1. Zunächst wird die verwendete Division mit Rest dahingehend abgeändert, daß sie anstatt des kleinsten positiven Rests den betragsmäßig kleinsten Rest zurückliefert.

2. Desweiteren reduzieren sie mit dem aktuellen Pivotelement alle Elemente der aktuellen Arbeitszeile und nicht nur ein Element. Dies hat den Vorteil, daß nach einem Reduktionsdurchlauf alle Elemente der Zeile betragsmäßig kleiner als das aktuelle Pivotelement sind.

Der Algorithmus hat dann mit diesen Modifikationen das folgende Aussehen:

10.23. Algorithmus

Modifikation von Havas und Majewski: $mgcd_{BestRemainder}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$
AUSGABE: $T \in \mathrm{GL}_n(\mathbb{Z})$ mit $a_{m-1,*} \cdot T = (0, \ldots, 0, \gcd(a_{m-1,*}))$.

```
     [Initialisierung]
(1)    T = I_n;
     [mgcd–Algorithmus]
(2)    j = index_abs_min(v);                 //|v_j| = min(|v_0|, ..., |v_{n-1}|)
(3)    swap(v_j, v_{n-1});                   //v_j <--> v_{n-1}
(4)    T.swap_columns(n − 1, j);
(5)    while (∃l : v_l ≠ 0 ∧ l ≠ n − 1) do
(6)      for (i = 0; i < n; i++) do
(7)        best_div_rem(q, r, v_i, v_{n-1}); //v_i = q · v_{n-1} + r
(8)        v_i = v_i − q · v_{n-1};
(9)        t_{*,i} = t_{*,i} − q · t_{*,n-1};
(10)       j = index_abs_min(v);             //|v_j| = min(|v_0|, ..., |v_{n-1}|)
(11)       swap(v_j, v_{n-1});               //v_j <--> v_{n-1}
(12)       T.swap_columns(n − 1, j);
(13)     od
(14)   od
(15)   return (T);
```

Auf die theoretische Seite der Modifikationen wollen wir an dieser Stelle nicht näher eingehen, sondern verweisen auf [MH94b, HHR93, HM94b].

Ergebnisse:
Auf der praktischen Seite haben diese relativ einfachen Modifikationen des Algorithmus von Blankinship einen großen Einfluß auf die von uns betrachteten Kenngrößen. Dies kann man aus den Tabellen 10.22, 10.23, 10.24 und 10.25 ersehen.

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	0.044
Bereich (bis)	3.8	1509.1	5554.8	60255.8	7847.65	2.114
abhängig von $L_0(v)$	√	√	√	√	√	√
abhängig von $L_\infty(A)$	√	√	√	√	√	√
abhängig von Eintragsverteilung von A	–	–	–	–	–	–

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	1	994.3	1	99	25.4873	1.045
450.1	90.02	99	1	944.6	1	99	23.6107	0.943
403.9	80.78	99	1	898.2	1	99	22.834	0.848
350.7	70.14	99	1	846.3	1	99	21.1815	0.741
295.6	59.12	98.9	1	791.8	1	98.9	18.9975	0.631
253.4	50.68	98.8	1.1	763	3.8	96.8	16.1556	0.568
198.7	39.74	98.7	1.4	735.8	15.8	94.2	12.2527	0.513
156.3	31.26	98.6	1.3	681.4	6.5	89	10.2785	0.399
97	19.4	98.7	1.3	610.2	7.4	90.4	7.65241	0.255
51.6	10.32	98.1	1.9	579.2	20.7	74.4	3.68612	0.197
29.5	5.9	97.1	1.8	542	19.7	81	2.9297	0.126
11	2.2	93.6	2.4	520.6	10.1	45.4	1.54264	0.084
5.4	1.08	82.9	2.2	508.2	8.9	55.4	1.39728	0.063
2.8	0.56	62.7	2	503.8	9.4	36.3	1.15522	0.057
1.8	0.36	57.3	1.7	502	2.9	22.2	1.05976	0.048
0.5	0.1000000	18.8	1	500	1	1	1	0.044

Tabelle 10.22: $mgcd_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	1.4	1113.9	102.6	918.8	193.613	1.309
449	89.8	997.6	1.5	1078.5	128.8	910.5	170.849	1.235
398.2	79.64	996	1.7	1093.5	125.1	753.2	117.307	1.256
348.9	69.78	996.6	1.7	1006.6	113.7	771.5	114.979	1.081
303.2	60.64	996.2	1.8	974.2	125.8	741.4	96.5735	1.012
243.8	48.76	995.1	1.8	898.2	92.1	766.2	92.9849	0.851
197.8	39.56	992.8	1.7	812.5	54.5	699.9	79.2246	0.665
149	29.8	992.9	1.9	746.6	119.8	679.3	60.3799	0.533
98	19.6	987.6	1.9	667	60.1	578.5	37.3984	0.371
48.4	9.68	978.9	2	586.1	57.6	536.3	21.8038	0.212
25.7	5.14	962.9	2.7	557.1	97.4	361	8.86053	0.157
8.6	1.72	828.9	2.5	518	107.9	477.9	5.08745	0.08
6.2	1.24	887.6	3.2	515.3	80.9	365.7	3.58587	0.081
2.6	0.52	624.3	2.2	504.1	46.4	309.7	2.29677	0.057
2	0.4	567.1	2	503.3	58.8	207.1	1.8025	0.059
1.1	0.22	257.8	1.2	500.8	49.8	103.9	1.44688	0.049

Tabelle 10.23: $mgcd_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2	1465	450.8	6764.4	1076.19	2.023
455.2	91.04	9985	2	1316.7	895.8	5626.6	794.381	1.718
397.3	79.46	9961.8	1.9	1234.9	184	4914.9	729.653	1.546
352.3	70.46	9977.6	1.9	1135.4	500	6714.6	978.285	1.345
301.7	60.34	9968.3	2	1071.9	413.4	4292	548.589	1.212
252.5	50.5	9945.4	1.9	952.1	533.1	4805.7	587.13	0.961
197.8	39.56	9954.8	2.1	894.5	245.2	5007.9	544.15	0.842
151.5	30.3	9926	2.1	799.4	574.3	4504.3	366.135	0.642
99.9	19.98	9907.7	2.2	706.9	264.8	4559.2	295.639	0.451
51.9	10.38	9677.3	2.5	614.3	452	4151.4	164.544	0.264
26.4	5.28	9739.2	3.2	575.6	517.2	3124.3	54.6499	0.193
12.8	2.56	9122.1	3.2	535	309.4	2242.3	30.2213	0.115
4.2	0.84	8115.2	3.2	510.3	680.6	2904.4	14.3079	0.071
3.9	0.78	8317.4	2.9	508.4	396.1	2393.6	13.1635	0.073
1.7	0.34	4758.2	1.8	503.5	322.6	1073.2	5.97994	0.056
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.046

Tabelle 10.24: $mgcd_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.1	1509.1	3719.2	41344.3	6097.64	2.114
452.1	90.42	99823.7	2	1396.5	516.9	44075.4	7122.99	1.878
399.3	79.86	99763.5	2.2	1336.2	1660.8	40046.6	5680.13	1.756
352.9	70.58	99657.1	2.2	1242.1	3731.6	60255.8	7847.65	1.582
307.4	61.48	99466.2	2.2	1158.7	921.3	57331.3	7464.8	1.388
251.8	50.36	99431.8	2.3	1046.7	5554.8	47066	4722.69	1.152
203.4	40.68	99511.4	2.9	1040.5	2770.5	21855.1	1549.47	1.141
150.7	30.14	99297.6	2.6	878.3	1956.2	24327.2	1814.13	0.802
104.3	20.86	98793.9	2.8	770.2	3286.1	21243.7	1147.31	0.575
54.8	10.96	97747.7	3.2	663.4	2399.7	23189.4	883.357	0.367
24.6	4.92	96337.4	3.5	578.2	2165.7	14825.4	267.281	0.205
11.5	2.3	92767.5	3.5	535.7	3170.4	17922.6	178.439	0.123
5.1	1.02	79355.4	3.8	515.8	1028.6	4943.2	36.3447	0.087
2.3	0.46	58436	2.3	504.9	4927.1	16413.7	61.8214	0.064
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.048
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.051

Tabelle 10.25: $mgcd_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

1. **Anzahl der Nicht–Nulleinträge des Darstellungsvektors x ($L_0(x)$)**
   Der größte gemeinsame Teiler kann als Linearkombination von 1, 2, 3 oder maximal 4 Einträgen des Eingabevektors dargestellt werden. Desweiteren ist eine leichte Abhängigkeit zwischen der Anzahl der zur Darstellung des größten gemeinsamen Teilers benötigten Einträge und $L_\infty(A)$ erkennbar.

2. **Anzahl der Nicht–Nulleinträge der Transformationsmatrix T ($L_0(T)$)**
   Die Anzahl der Nicht–Nulleinträge der Transformationsmatrix T hängt wiederum direkt an der Anzahl der zur Darstellung des größten, gemeinsamen Teilers benötigten Elemente. Somit besteht eine leichte Abhängigkeit zwischen der Anzahl der Nicht–Nullelemente und der Größe der Einträge der Startmatrix.

3. **maximaler Eintrag des Darstellungsvektors x ($L_\infty(x)$)**
   Die Größe des maximalen Eintrags des Darstellungsvektors ist deutlich kleiner im Vergleich zu der entsprechenden Kenngröße des Modulalgorithmus $mgcd_{Blankinship}$ und im Schnitt sogar besser als der maximale Eintrag des Modulalgorithmus $mgcd_{Bradley}$. Es besteht ebenfalls eine leichte Abhängigkeit der Größe des maximalen Eintrags des Darstellungsvektors von der Schranke der Eintragsgröße der Startmatrix. $L_\infty(A)$ ist eine obere Schranke für den maximalen Eintrag.

4. **maximaler Eintrag der Transformationsmatrix T ($L_\infty(T)$)**
   Die maximalen Einträge der Transformationsmatrix übersteigen bei diesem Modulalgorithmus die jeweiligen maximalen Einträge des Darstellungsvektors. Aber $L_\infty(A)$ stellt immer noch eine obere Schranke für den maximalen Eintrag dar. Die Abhängigkeiten zwischen der Größe des maximalen Eintrags des Darstellungsvektors und der Eintragsdichte der Startmatrix bzw. der Größe der Einträge der Startmatrix übertragen sich eins zu eins auf den maximalen Eintrag der Transformationsmatrix. Im Vergleich zum Modulalgorithmus $mgcd_{Blankinship}$ schneidet dieser Modulalgorithmus bzgl. der hier betrachteten Kenngröße deutlich besser ab, fällt aber leicht hinter den Modulalgorithmus $mgcd_{Bradley}$ zurück.

5. **durchschnittliche Eintragsgröße der Transformationsmatrix T ($\Delta_\infty(T)$)**
   Die durchschnittliche Eintragsgröße der Nicht–Nulleinträge der Transformationsmatrix übersteigt wie der maximale Eintrag der Transformationsmatrix deutlich den maximalen Eintrag des Darstellungsvektors, bleibt aber deutlich hinter dem maximalen Eintrag der Transformationsmatrix zurück. Die oben erwähnten Abhängigkeiten übertragen sich eins zu eins auf die hier betrachtete Kenngröße.

6. **Laufzeit (t)**
   Die Laufzeiten bewegen sich zwischen 0,044 bis 2,114 CPU–Sekunden. Damit ist dies der bisher schnellste Modulalgorithmus. Es besteht eine Abhängigkeit von der Größe der Einträge der Startmatrix. Eine direkte Abhängigkeit zwischen der Laufzeit und der Eintragsdichte ist ebenfalls zu erkennen.

Pivotierung

Ein weitere Modifikation des Algorithmus von Blankinship liegt darin, durch eine geeignete Pivotstrategie eine weitere Verbesserung in Bezug auf die Größe der entstehenden Zwischeneinträge und damit in Bezug auf das Laufzeitverhalten zu erreichen. Dabei wird die Auswahl eines Pivotelementes durch den Bereich, in dem wir nach dem Pivotelement

suchen können, und durch ein Kriterium, das uns sagt, ob ein Element der Arbeitszeile als Pivot geeignet ist, beinflußt.

Pivotbereich:
Bei der Auswahl des Suchbereiches für die Pivotierung müssen wir folgende zwei Aspekte in Betracht ziehen:

- Je mehr Elemente wir zur Auswahl haben, desto größer ist die Wahrscheinlichkeit, ein Element mit guten Pivoteigenschaften zu finden.
- Je mehr Elemente wir zur Auswahl in Betracht ziehen, desto größer ist der Zeitbedarf für die Pivotierung.

Den „Trade of“ muß man in Abhängigkeit von den speziell vorliegenden Problemen und der Komplexität des jeweiligen Pivotkriteriums untersuchen. Bei den praktischen Tests, die wir durchgeführt haben, hat sich keine Notwendigkeit einer Beschränkung des Pivotbereiches ergeben. Deshalb wird im folgenden immer der maximale Pivotbereich verwendet.

Für die HNF–Berechnung umfaßt der Pivotbereich die Elemente links von der aktuellen Arbeitsposition in der gleichen Zeile.

Es stellt sich die Frage, was gute Pivoteigenschaften sind bzw. was das Kriterium ist, das uns ein Element als Pivotelement attraktiv macht.

Pivotkriterium:
Je nach Blickwinkel lassen sich unterschiedliche Kriterien für die Wahl eines geeigneten Pivotelementes ableiten:

1. Betrachtet man die Pivotwahl aus dem Blickwinkel der Ausführungseffizienz, sind die Einheiten von $\mathbb{Z}$ von besonderer Bedeutung. Gelingt es uns, als Pivot eine Einheit zu finden, ist die mgcd–Berechnung im Algorithmus zur HNF–Berechnung besonders einfach und die Elimination der übrigen Elemente der Pivotzeile kann sehr effizient ausgeführt werden. Kleine Pivotelemente haben den Vorteil, daß nur wenige Durchläufe benötigt werden, um den gcd der Elemente der Pivotzeile zu bestimmen und die Elimination durchzuführen [HS79]. Dem entsprechend lautet unsere erste Pivotstrategie:

   Pivot$_1$

Wähle als Pivotelement das betragsmäßig kleinste Element des Pivotbereiches!

   Diese Pivotstrategie ist bereits im Modulalgorithmus $mgcd_{BestRemainder}$ enthalten. Eine Ergebnisanalyse entfällt somit. Im weiteren Verlauf dieser Arbeit sehen wir die Modulalgorithmusbezeichungen $mgcd_{BestRemainder}$ und $mgcd^{Pivot_1}_{BestRemainder}$ als synonym an.

2. Betrachten wir nun das Problem der Pivotwahl aus dem Blickwinkel der Bekämpfung der *entry explosion*. In der mgcd–Berechnung ist, wie man sich leicht an den oben vorgestellten Algorithmen klarmachen kann, implizit eine Elimination der übrigen

Elemente der Pivotzeile enthalten.

In dieser Eliminationsphase wird ein der Darstellung des mgcd entsprechendes Vielfaches der Pivotspalte zu den anderen Spalten addiert, um die Elemente in der Pivotzeile zu eliminieren. Dies führt folgenden Überlegungen:

(a) In der Pivotierung sollte man demjenigen Element den Vorzug geben, das zu der oben beschriebenen Addition am wenigsten beiträgt, d.h. dessen Spaltennorm gemäß einer vorgegebenen Norm die kleinste unter den beteiligten Elementen ist. Dies legt die Idee nahe, in das Pivotkriterium der HNF–Berechnung die Spaltennorm des jeweiligen Elementes zu integrieren. Als Spaltennorm verwenden wir die L_k–Norm ($k > 0$). Für $k = 1$ werden die Beträge der Elemente der jeweiligen Spalte addiert und für $k = 2$ erhalten wir die euklidische Norm.

Als Pivotkriterium erhalten wir nach dieser Betrachtung:

Pivot$_2$

Wähle als Pivot ein Element ungleich Null des Pivotbereiches, dessen Spaltennorm minimal ist und das nicht maximalen Betrag unter den Elementen des Pivotbereiches besitzt!

Ergebnisse:
Dies führt zu den Ergebnissen, die in den Tabellen 10.26, 10.27, 10.28, 10.29 für die Maximumnorm und in den Tabellen 10.30, 10.31, 10.32, 10.33 für die euklidische Norm zusammengestellt wurden.
Betrachten wir zuerst die erzielten Ergebnis für die Maximumnorm ($k = 1$):

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	0.044
Bereich (bis)	6.3	3232	2179.4	9952.4	1012.96	7.223
abhängig von $L_0(v)$	√	√	√	√	√	√
abhängig von $L_\infty(A)$	√	√	√	√	√	√
abhängig von Eintragsverteilung von A	√	√	√	√	√	√

i. **Anzahl der Nicht–Nulleinträge des Darstellungsvektors** x **($L_0(x)$)**
Dieser Modulalgorithmus verhält sich in Bezug auf diese Kenngröße schlechter als der Modulalgorithmus $mgcd_{BestRemainder}$. In Abhängigkeit von der Größe der Einträge der Startmatrix werden zur Darstellung des größten gemeinsamen Teilers 1 bis 2 bei $L_\infty(A) < 100$ bzw. 4 bis 7 bei $L_\infty(A) < 100000$ benötigt. Eine direkte Abhängigkeit zur Eintragsdichte der Startmatrix ist erkennbar.

ii. **Anzahl der Nicht–Nulleinträge der Transformationsmatrix** T **($L_0(T)$)**
Entsprechend dem Verhalten der Kenngröße $L_0(x)$ verhält sich die hier betrachtete Kenngröße. Im Vergleich zu dem Modulalgorithmus $mgcd_{BestRemainder}$ produziert dieser Modulalgorithmus Transformationsmatrizen mit einer deutlich höheren Anzahl an Nicht–Nulleinträgen. Die für $L_0(x)$ beschriebenen Abhängigkeiten übertragen sich eins zu eins auf die hier betrachtete Kenngröße.

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2	1798.9	5.3	37.1	6.50342	3.362
450.1	90.02	99	1.9	1665.2	5	35.1	6.01742	3.049
403.9	80.78	99	2	1520	8	37.8	7.15507	2.716
350.7	70.14	99	1.9	1403	3.1	38.2	6.3784	2.364
295.6	59.12	98.9	2	1215.8	16.7	46	6.74963	1.841
253.4	50.68	98.8	2.1	1077.3	10.7	52.1	8.85102	1.57
198.7	39.74	98.7	2.1	992.3	4.8	46.3	6.66522	1.256
156.3	31.26	98.6	2.1	921.6	6.8	34.2	4.77094	1.099
97	19.4	98.7	2	736	6.7	36.6	3.55978	0.637
51.6	10.32	98.1	2.5	627.9	3.6	35.3	3.22663	0.329
29.5	5.9	97.1	2.7	578.8	7.4	33.3	2.1764	0.22
11	2.2	93.6	3.1	528.1	4.3	28	1.46582	0.111
5.4	1.08	82.9	2.6	510.3	14.6	52	1.44876	0.072
2.8	0.56	62.7	2	504.6	8.5	29.7	1.13099	0.06
1.8	0.36	57.3	1.8	502.1	6	21.1	1.06811	0.051
0.5	0.1000000	18.8	1	500	1	1	1	0.044

Tabelle 10.26: $mgcd^{Pivot_2}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.4	2209.9	26	247.7	34.4333	4.505
449	89.8	997.6	2.5	2254.1	5.6	62.3	6.70769	4.617
398.2	79.64	996	2.4	1981.2	2.5	56.4	9.45134	3.975
348.9	69.78	996.6	2.4	1648.4	13.9	169.7	29.2583	3.21
303.2	60.64	996.2	2.9	1635.9	4.8	90	14.5302	3.065
243.8	48.76	995.1	3.2	1421.1	14.3	116.4	17.5916	2.388
197.8	39.56	992.8	2.9	1244.9	15.8	152.8	20.1573	1.973
149	29.8	992.9	2.5	969.4	5.8	147.2	18.2803	1.224
98	19.6	987.6	2.8	825.2	9	192.7	15.2665	0.865
48.4	9.68	978.9	2.8	668	11.1	151.5	9.0774	0.422
25.7	5.14	962.9	3.4	576.5	25	176.2	6.95733	0.225
8.6	1.72	828.9	3.5	526.4	46.9	250.6	2.99145	0.103
6.2	1.24	887.6	3.6	518	72.5	219.6	2.7888	0.093
2.6	0.52	624.3	2.4	504.3	45	290.9	2.24093	0.063
2	0.4	567.1	1.9	503	88.2	282.3	2.2497	0.058
1.1	0.22	257.8	1.4	501.2	12.4	25.7	1.19453	0.052

Tabelle 10.27: $mgcd^{Pivot_2}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	3.3	2837.7	4.7	488.3	64.358	6.188
455.2	91.04	9985	3.3	2552.5	5.4	275.6	30.8352	5.545
397.3	79.46	9961.8	3.5	2116.7	191.3	955.5	98.4669	4.306
352.3	70.46	9977.6	3.6	2073.7	6.2	227.2	30.1844	4.161
301.7	60.34	9968.3	3.3	1844.5	3.1	567.7	73.1039	3.536
252.5	50.5	9945.4	4.2	1699.1	47.4	372.1	35.2389	3.185
197.8	39.56	9954.8	4.9	1395.3	9.9	240.1	25.6489	2.363
151.5	30.3	9926	4.2	1213.1	13.6	205.6	20.6342	1.898
99.9	19.98	9907.7	4.6	1001	8.6	212	14.8139	1.215
51.9	10.38	9677.3	4.4	717.4	37.4	318.7	16.6543	0.577
26.4	5.28	9739.2	4.6	620.3	21	257	9.19233	0.323
12.8	2.56	9122.1	3.9	551.7	39.8	514.3	6.49375	0.178
4.2	0.84	8115.2	3.6	512.6	479.6	1758.4	10.1797	0.087
3.9	0.78	8317.4	3	509.6	321.8	1366	7.34125	0.084
1.7	0.34	4758.2	1.7	503	371.4	1554.7	9.75388	0.055
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.05

Tabelle 10.28: $mgcd^{Pivot_2}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	4.5	3232	12.4	835.9	86.3924	7.223
452.1	90.42	99823.7	4	2903.7	25.8	490.4	49.1691	6.389
399.3	79.86	99763.5	5.5	3042.2	5.3	239.9	19.6308	6.703
352.9	70.58	99657.1	5.5	2701.5	8	682.2	38.4588	5.34
307.4	61.48	99466.2	5	2194.3	99.7	980.3	80.876	4.626
251.8	50.36	99431.8	4.8	2015.2	17.5	289	25.6866	3.778
203.4	40.68	99511.4	5.2	1631.8	12.4	569.1	40.4026	3.025
150.7	30.14	99297.6	5.4	1340.6	16.6	6456.7	583.818	2.221
104.3	20.86	98793.9	5.5	1101.6	32.5	9952.4	1012.96	1.504
54.8	10.96	97747.7	6.3	838.8	19.7	339.2	22.6059	0.861
24.6	4.92	96337.4	5	615.9	63.4	1026.4	23.6365	0.347
11.5	2.3	92767.5	5.1	553.3	456.2	5654.2	77.2304	0.188
5.1	1.02	79355.4	4	517.4	1022.5	4040.7	22.7574	0.108
2.3	0.46	58436	2.3	504.9	1710	9141.5	31.3012	0.074
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.049
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.051

Tabelle 10.29: $mgcd^{Pivot_2}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

iii. **maximaler Eintrag des Darstellungsvektors** x **(**$L_\infty(x)$**)**
Die Größe des maximalen Eintrags des Darstellungsvektor ist deutlich kleiner im Vergleich zu der entsprechenden Kenngröße des Modulalgorithmus $mgcd_{BestRemainder}$. Eine Abhängigkeit von $L_\infty(A)$ bzw. der Eintragsdichte der Startmatrix ist nicht mehr erkennbar.

iv. **maximaler Eintrag der Transformationsmatrix** T **(**$L_\infty(T)$**)**
Analog zu $L_\infty(x)$ verhält sich die Kenngröße $L_\infty(T)$. Die maximalen Einträge der Transformationsmatrix übersteigen auch bei diesem Modulalgorithmus deutlich die jeweiligen maximalen Einträge des Darstellungsvektors. Im Vergleich zu dem Modulalgorithmus $mgcd_{BestRemainder}$ sind die Einträge deutlich kleiner und eine Abhängigkeit zwischen der Größe des maximalen Eintrags und der Eintragsdichte der Startmatrix bzw. der Größe der Einträge der Startmatrix ist nicht mehr erkennbar.

v. **durchschnittliche Eintragsgröße der Transformationsmatrix** T **(**$\Delta_\infty(T)$**)**
Die im vorangegangen Abschnitt zusammengestellten Beobachtungen übertragen sich direkt auf diese Kenngröße. Bemerkenswert ist, daß auch bei diesem Modulalgorithmus eine deutliche Differenz zwischen dem maximalen Eintrag einer Transformationsmatrix und der durchschnittlichen Eintragsgröße besteht.

vi. **Laufzeit (**t**)**
Die Laufzeiten hängen von der Größe der Einträge der Startmatrix ab. Sie bewegen sich zwischen 0.05 bis 3.362 CPU–Sekunden für $L_\infty(A) < 100$ und 0.051 bis 7.223 CPU–Sekunden für $L_\infty(A) < 100000$. Ein Einfluß der Eintragsdichte der Startmatrix auf die Laufzeit ist nicht erkennbar.

Für die euklidische Norm ergibt sich ein ähnliches Bild. Die Aussagen für die Kenngrößen $L_0(x)$ und $L_0(T)$ im Fall $k = 1$ gelten auch für den Fall $k = 2$. Für die restlichen Kenngrößen stellen wir folgendes fest:

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	0.046
Bereich (bis)	6.7	3702.5	2179.4	9816.3	33.974	15.007
abhängig von $L_0(v)$	$\surd$	$\surd$	$\surd$	$-$	$-$	$\surd$
abhängig von $L_\infty(A)$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
abhängig von Eintragsverteilung von A	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

i. **maximaler Eintrag des Darstellungsvektors** x **(**$L_\infty(x)$**)**
Die guten Ergebnisse für $k = 1$ bzgl. der Kenngröße $L_\infty(x)$ werden nochmals deutlich übertroffen. Der maximale Eintrag ist deutlch kleiner als im Fall $k = 1$.

ii. **maximaler Eintrag der Transformationsmatrix** T **(**$L_\infty(T)$**)**
Ebenso ergibt ich auch für die Kenngröße $L_\infty(T)$ eine nochmalige Verbesserung im Vergleich zum Fall $k = 1$.

iii. **durchschnittliche Eintragsgröße der Transformationsmatrix** T **(**$\Delta_\infty(T)$**)**
Die durchschnittliche Eintragsgröße orientiert sich wiederum an der maximalen Eintragsgröße. Sie ist somit ebenfalls deutlich kleiner als im Fall

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	1.1	1797.7	1	10.9	2.25905	6.249
450.1	90.02	99	1.1	1698.1	1	6	1.65997	6.145
403.9	80.78	99	1.1	1544.7	1	8	1.84592	5.361
350.7	70.14	99	1.1	1388	1	14.4	2.62932	4.418
295.6	59.12	98.9	1.1	1225.4	1	13.2	2.37767	3.2
253.4	50.68	98.8	1.2	1164.7	1	12.5	2.1785	3.038
198.7	39.74	98.7	1.5	1008.1	1	9.2	1.73574	2.141
156.3	31.26	98.6	1.6	906.5	1	9	1.61147	1.659
97	19.4	98.7	1.5	760.3	1	7.8	1.43825	0.993
51.6	10.32	98.1	1.9	628.8	1.1	9.2	1.36864	0.433
29.5	5.9	97.1	2.4	574.1	1.3	14.3	1.33827	0.26
11	2.2	93.6	3.3	531.7	1.8	13.2	1.16363	0.137
5.4	1.08	82.9	2.4	510.3	6.9	36.3	1.22222	0.081
2.8	0.56	62.7	2	503.8	8	27.9	1.10877	0.062
1.8	0.36	57.3	1.7	502.1	2.9	13.6	1.04242	0.056
0.5	0.1000000	18.8	1	500	1	1	1	0.046

Tabelle 10.30: $mgcd^{Pivot_2}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	1.6	2363.5	1	16.2	2.96074	9.452
449	89.8	997.6	1.8	2082.1	1	61.1	7.58556	7.739
398.2	79.64	996	1.9	1886.6	1.1	23.1	3.58163	6.716
348.9	69.78	996.6	2.1	1791.5	1	12.9	2.27117	6.383
303.2	60.64	996.2	2.3	1684.6	1	16.2	2.5187	5.545
243.8	48.76	995.1	2.5	1438.3	1	15	2.26517	4.323
197.8	39.56	992.8	2.3	1281.3	1	11.2	1.94225	3.535
149	29.8	992.9	2.6	1082.9	1.1	21	2.45166	2.42
98	19.6	987.6	2.8	865.3	1.4	45.8	3.93124	1.375
48.4	9.68	978.9	3.2	685.6	1.3	30.5	2.12763	0.619
25.7	5.14	962.9	3.9	604.6	1.9	14.3	1.37512	0.358
8.6	1.72	828.9	3.4	528.9	26.9	80.7	1.58272	0.134
6.2	1.24	887.6	4	522.9	35.6	124.2	1.84184	0.119
2.6	0.52	624.3	2.5	505.1	43.3	271	2.06296	0.069
2	0.4	567.1	2	503.3	55.2	232.9	1.82535	0.065
1.1	0.22	257.8	1.2	500.8	29.2	105.7	1.49141	0.056

Tabelle 10.31: $mgcd^{Pivot_2}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	3.2	3246.2	1	10.1	1.99473	13.494
455.2	91.04	9985	3.2	2906.5	1.1	32.2	3.92076	11.971
397.3	79.46	9961.8	3.2	2338.7	1.2	28.1	4.88677	7.993
352.3	70.46	9977.6	3.4	2293.8	1.1	33.3	3.6736	8.345
301.7	60.34	9968.3	2.8	2116.5	1.2	21.6	2.97812	7.705
252.5	50.5	9945.4	4.4	1868.6	1	21.3	2.73472	5.973
197.8	39.56	9954.8	4.5	1568.7	1.3	18.3	2.35832	4.575
151.5	30.3	9926	4.4	1311.7	1.6	40.3	3.45391	3.345
99.9	19.98	9907.7	4.8	1028.2	1.6	30.2	3.49144	1.952
51.9	10.38	9677.3	4.4	773.2	1.7	21.2	2.1367	0.926
26.4	5.28	9739.2	4.9	637.6	3.7	31.1	2.14931	0.454
12.8	2.56	9122.1	5.5	565.7	5.6	39.7	1.85346	0.236
4.2	0.84	8115.2	3.6	512.2	403.5	1010.6	5.10953	0.096
3.9	0.78	8317.4	3.6	511.3	92.3	607.2	3.52494	0.100
1.7	0.34	4758.2	1.8	503.8	321.2	996.1	5.46427	0.063
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.053

Tabelle 10.32: $mgcd^{Pivot_2}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	4	3702.5	1.1	41.7	5.31757	15.007
452.1	90.42	99823.7	4	3315.5	1.4	90.2	7.79554	13.016
399.3	79.86	99763.5	4.8	3001.3	2.8	93.6	7.78733	11.812
352.9	70.58	99657.1	4.2	2861.1	4	120.3	9.37982	10.499
307.4	61.48	99466.2	4.6	2467.7	2.2	29.6	3.83685	9.375
251.8	50.36	99431.8	4.9	2264	1.5	47.5	5.38891	8.043
203.4	40.68	99511.4	5.5	1887.6	1.3	104.8	11.464	5.815
150.7	30.14	99297.6	5.5	1479.9	2.2	26.9	3.70599	4.042
104.3	20.86	98793.9	5.5	1232.7	1.7	26.8	3.02061	2.74
54.8	10.96	97747.7	6.7	935.5	2	19	2.30722	1.405
24.6	4.92	96337.4	6.7	658	4.9	46.6	2.75152	0.531
11.5	2.3	92767.5	5.5	563.8	29.1	189.6	3.47251	0.238
5.1	1.02	79355.4	4	518	77.4	601.3	4.86081	0.125
2.3	0.46	58436	2.4	505.5	1676.6	7274.2	26.4477	0.081
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.051
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.05

Tabelle 10.33: $mgcd^{Pivot_2}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$k = 1$. Es besteht weiterhin eine deutliche Differenz zwischen dem maximalen Eintrag einer Transformationsmatrix und der durchschnittlichen Eintragsgröße.

iv. **Laufzeit (t)**
Die Laufzeiten hängen wie im Fall $k = 1$ von der Größe der Einträge der Startmatrix ab. Sie bewegen sich zwischen 0.046 bis 6.249 CPU–Sekunden für $L_\infty(A) < 100$ und 0.05 bis 15.007 CPU–Sekunden für $L_\infty(A) < 100000$. Offensichtlich bezahlen wir die oben dargestellten Verbesserungen mit einer erhöhten Laufzeit.

(b) In der ersten Überlegung liegt der Schwerpunkt der Betrachtung auf der Größe der Elemente der jeweiligen Spalte. Im worst–case entspricht dies bei der Verarbeitung der Spalten i und j

$$\max_k\{|a_{k,i} - q \cdot a_{k,j}|\} = \max_k\{|a_{k,j}|\} + |q| \cdot \max_k\{|a_{k,i}|\}$$

Auf der anderen Seite kann man auch den Koeffizienten q betrachten, der aus der Division mit Rest des Pivotelements mit einem anderen Element entsteht. Wählt man das Pivotelement derart, daß dieser Koeffizient klein ist, führt dies ebenfalls zu einer Minimierung der Größe der entstehenden Zwischeneinträge [HMM94].

Als Pivotelement ergibt sich somit:

Pivot$_3$

Wähle als Pivotelement das betragsmäßig zweitgrößte Element des Pivotbereichs!

Der Algorithmus, der bei Verwendung dieser Pivotstrategie entsteht, trägt den Namen "sorting gcd method" und wurde im Detail in [MH95] untersucht.

10.24. Bemerkung
Die Reduktion aller möglichen Elemente, d.h. aller Elemente deren Betrag größer oder gleich dem Betrag des Pivotelements ist, bringt einen weiteren Geschwindigkeitsvorteil.

Ergebnisse:
Der sich mittels der Pivotstrategie $Pivot_3$ ergebende Modulalgorithmus führt zu den in den Tabellen 10.34, 10.35, 10.36 und 10.37 zusammengestellten Ergebnissen.

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	0.047
Bereich (bis)	14.5	3716.9	2179.4	9816.3	33.974	5.059
abhängig von $L_0(v)$	√	√	√	−	−	√
abhängig von $L_\infty(A)$	√	√	√	√	√	√
abhängig von Eintragsverteilung von A	√	√	√	√	√	√

(a) **Anzahl der Nicht–Nulleinträge des Darstellungsvektors x ($L_0(x)$)**
Bei diesem Modulalgorithmus kann der größte gemeinsame Teiler als Linearkombination von 2 bis 3 für $L_\infty(A) < 100$ bzw. 4 bis 7 für $L_\infty(A) < 100000$

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2.6	1452.9	1	2	1.0139	2.15
450.1	90.02	99	2.6	1372.6	1	2.3	1.01552	1.966
403.9	80.78	99	2.3	1294.1	1	2	1.01035	1.796
350.7	70.14	99	2.7	1225	1	2.2	1.02041	1.597
295.6	59.12	98.9	2.6	1135.8	1	2.4	1.02051	1.398
253.4	50.68	98.8	2.1	1056.4	1	2.1	1.0088	1.224
198.7	39.74	98.7	2.5	972.1	1	2.2	1.01543	1.022
156.3	31.26	98.6	2.6	905.4	1	2.4	1.01833	0.819
97	19.4	98.7	3.1	785.4	1	2.6	1.01999	0.563
51.6	10.32	98.1	3.3	675.1	1	2.7	1.02266	0.339
29.5	5.9	97.1	3.4	605.4	1.1	2.5	1.02494	0.227
11	2.2	93.6	5.2	547.4	2.3	7.9	1.08221	0.125
5.4	1.08	82.9	3.7	516.8	6.3	20.8	1.13777	0.088
2.8	0.56	62.7	2.3	505.4	9.1	30.2	1.11694	0.06
1.8	0.36	57.3	1.7	502.2	2.6	13.4	1.0462	0.057
0.5	0.1000000	18.8	1	500	1	1	1	0.047

Tabelle 10.34: $mgcd^{Pivot_3}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.8	2172	1	2.3	1.01611	3.435
449	89.8	997.6	4	2128.8	1.1	2.9	1.03514	3.071
398.2	79.64	996	3.9	1959.1	1.1	2.9	1.03787	2.761
348.9	69.78	996.6	4.4	1813.4	1	2.6	1.02873	2.44
303.2	60.64	996.2	4.1	1633.6	1.1	3.1	1.03838	2.15
243.8	48.76	995.1	3.7	1429.1	1.1	3	1.03408	1.73
197.8	39.56	992.8	3.6	1261.5	1.1	2.6	1.02822	1.425
149	29.8	992.9	4.5	1131.8	1.3	3.2	1.03949	1.092
98	19.6	987.6	5	971.1	1.4	3.7	1.0485	0.753
48.4	9.68	978.9	5.2	779.1	1.6	4.1	1.06084	0.435
25.7	5.14	962.9	7.5	685.8	1.7	4.6	1.07437	0.273
8.6	1.72	828.9	6.1	546.7	2.9	8	1.11304	0.132
6.2	1.24	887.6	5.1	529.7	34.4	85.7	1.47631	0.112
2.6	0.52	624.3	2.6	505.3	46.4	244.6	1.91094	0.069
2	0.4	567.1	1.9	503.1	71.5	222.1	1.85669	0.065
1.1	0.22	257.8	1.4	501.4	5	13.9	1.05943	0.052

Tabelle 10.35: $mgcd^{Pivot_3}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	4.3	2591.8	1.1	2.9	1.03376	4.087
455.2	91.04	9985	4.8	2477.8	1	2.8	1.03273	3.727
397.3	79.46	9961.8	4.7	2315.7	1.1	3	1.03766	3.278
352.3	70.46	9977.6	4.9	2131.5	1	3	1.03167	2.961
301.7	60.34	9968.3	4.8	1970.5	1.1	3	1.03816	2.571
252.5	50.5	9945.4	4.9	1813.6	1	3	1.03667	2.204
197.8	39.56	9954.8	4.2	1570.2	1	3	1.0321	1.779
151.5	30.3	9926	5.3	1457.5	1.2	3.7	1.0483	1.41
99.9	19.98	9907.7	6.3	1243.6	1.3	3.9	1.05002	0.994
51.9	10.38	9677.3	8.8	974.1	1.7	5	1.07648	0.576
26.4	5.28	9739.2	9.8	775.2	2.4	6.1	1.12758	0.347
12.8	2.56	9122.1	9.4	612.5	3.5	13.2	1.26531	0.205
4.2	0.84	8115.2	4.1	515.5	405.3	964.3	4.7579	0.103
3.9	0.78	8317.4	3.7	512.6	384.9	1441.8	9.93894	0.102
1.7	0.34	4758.2	2	504.1	315.1	954.7	5.21762	0.067
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.053

Tabelle 10.36: $mgcd^{Pivot_3}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	5.3	3660.6	1.2	3.8	1.04718	5.059
452.1	90.42	99823.7	6.9	3716.9	1.5	4.2	1.05462	4.632
399.3	79.86	99763.5	5.3	3193.9	1.2	3.9	1.04737	4.158
352.9	70.58	99657.1	5.8	3046.3	1.1	4	1.04816	3.813
307.4	61.48	99466.2	6.7	2830.9	1.3	4	1.04013	3.295
251.8	50.36	99431.8	7.3	2547.6	1.2	4	1.04483	2.746
203.4	40.68	99511.4	8.6	2289.7	1.5	4.6	1.05813	2.251
150.7	30.14	99297.6	9	1968.6	1.4	4.3	1.05705	1.71
104.3	20.86	98793.9	9.8	1637.3	2.1	5.8	1.0921	1.238
54.8	10.96	97747.7	10.8	1225.7	2	6.6	1.12156	0.724
24.6	4.92	96337.4	14.5	852.5	4.5	13.3	1.3512	0.398
11.5	2.3	92767.5	9.6	609.9	9.2	23.5	1.52861	0.227
5.1	1.02	79355.4	4.9	522.7	1073.2	2567.1	13.6292	0.13
2.3	0.46	58436	2.5	506	1638.9	7200.6	24.6951	0.082
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.051
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.052

Tabelle 10.37: $mgcd^{Pivot_3}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

dargestellt werden. Dies bedeutet, es besteht eine Abhängigkeit zwischen der Größe des maximalen Eintrags der Startmatrix und von $L_0(x)$, Eine Abhängigkeit von der Eintragsdichte der Startmatrix ist nicht zu erkennen.

(b) **Anzahl der Nicht–Nulleinträge der Transformationsmatrix T ($L_0(T)$)**
Die Anzahl der Nicht–Nulleinträge der Transformationsmatrix T liegt leicht über der des Modulalgorithmus $mgcd^{Pivot_2}_{BestRemainder}(k = 2)$ und deutlich über der des Modulalgorithmus $mgcd_{BestRemainder}$.

(c) **maximaler Eintrag des Darstellungsvektors x ($L_\infty(x)$)**
Im Vergleich zu dem Modulalgorithmus $mgcd^{Pivot_2}_{BestRemainder}$ kann man eine leichte Verbesserung feststellen, d.h. der maximale Eintrag des Darstellungsvektors ist nochmals etwas kleiner geworden. Etwaige Abhängigkeiten zu der Eintragsdichte oder der Größe der Einträge der Startmatrix sind nicht erkennbar.

(d) **maximaler Eintrag der Transformationsmatrix T ($L_\infty(T)$)**
Der maximale Eintrag der Transformationsmatrix bei diesem Modulalgorithmus ist im Vergleich zu dem Modulalgorithmus $mgcd^{Pivot_2}_{BestRemainder}(k = 2)$ deutlich kleiner. Eine Abhängigkeit zur Eintragsdichte der Startmatrix bzw. der Größe der Einträge der Startmatrix ist nicht mehr erkennbar.

(e) **durchschnittliche Eintragsgröße der Transformationsmatrix T ($\Delta_\infty(T)$)**
Die durchschnittliche Eintragsgröße ist ebenfalls deutlich kleiner als für den Modulalgorithmus $mgcd^{Pivot_2}_{BestRemainder}(k = 2)$. Weiterhin ist absolute und relative Differenz zwischen dem maximalen Eintrag einer Transformationsmatrix und der durchschnittlichen Eintragsgröße deutlich kleiner geworden.

(f) **Laufzeit (t)**
Für den hier betrachteten Modulalgorithmus bewegen sich die Laufzeiten zwischen 0.047 und 2.15 CPU–Sekunden für $L_\infty(A) < 100$ und zwischen 0.052 und 5.059 CPU–Sekunden für $L_\infty(A) < 100000$. Es besteht somit eine direkte Abhängigkeit zwischen der Laufzeit und der Größe der Einträge der Startmatrix.

3. Aus dem Blickwinkel der Minimierung der Anzahl der neu entstehenden Einträge bietet sich eine analoge Vorgehensweise an. Anstatt mittels Spaltennormen zu versuchen, die Größe der entstehenden Zwischeneinträge zu minimieren, versuchen wir nun mittels der L_0–Metrik die Anzahl der neu entstehenden Einträge klein zu halten.

Als Pivotkriterium erhalten wir somit:

Pivot$_4$

Wähle als Pivot dasjenige Element ungleich Null des Pivotbereiches, dessen Spalte die minimale Anzahl an Elementen hat und das nicht maximalen Betrag unter den Elementen des Pivotbereiches besitzt!

10.25. Bemerkung
Rose und Tarjan haben gezeigt, das es ein NP-vollständiges Problem ist, eine Strategie zu finden, die die Anzahl der Fill-Ins minimiert [HHR93].

Ergebnisse:
Dieser Modulalgorithmus führt zu den Ergebnissen der Tabellen 10.38, 10.39, 10.40 und 10.41.

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	0.045
Bereich (bis)	6.2	3159.2	12918.5	43541.7	313.683	8.3
abhängig von $L_0(v)$	√	√	√	–	–	√
abhängig von $L_\infty(A)$	√	√	√	√	√	√
abhängig von Eintragsverteilung von A	√	√	√	√	√	√

(a) **Anzahl der Nicht–Nulleinträge des Darstellungsvektors x ($L_0(x)$)**
Dieser Modulalgorithmus führt zu Darstellungen des größten, gemeinsamen Teilers, die mehr Einträge benötigen als der Modulalgorithmus $mgcd_{BestRemainder}$. Die Anzahl der benötigten Elemente liegt in der Größenordnung des Modulalgorithmus $mgcd^{Pivot_2}_{BestRemainder}$.

(b) **Anzahl der Nicht–Nulleinträge der Transformationsmatrix T ($L_0(T)$)**
Für die Kenngröße $L_0(T)$ ist die Situation ähnlich. Die Anzahl der Nicht–Nulleinträge der Transformationsmatrix ist höher als die des Modulalgorithmus $mgcd_{BestRemainder}$.

(c) **maximaler Eintrag des Darstellungsvektors x ($L_\infty(x)$)**
Die Größe des maximalen Eintrags des Darstellungsvektor ist kleiner im Vergleich zu der entsprechenden Kenngröße des Modulalgorithmus $mgcd_{BestRemainder}$, aber deutlich schlechter im Vergleich zu den letzten drei vorgestellten Strategien bzw. Pivottechniken. Eine Abhängigkeit zu der Eintragsgröße der Einträge der Startmatrix ist deutlich zu erkennen, während wiederum eine Abhängigkeit zu der Eintragsdichte der Startmatrix nicht feststellbar ist.

(d) **maximaler Eintrag der Transformationsmatrix T ($L_\infty(T)$)**
Die Bemerkungen zu der Kenngröße $L_\infty(x)$ übertragen sich sinngemäß auf diese Kenngröße. Zusätzlich gilt, daß die Größe des maximalen Eintrags der Transformationsmatrix deutlich den maximalen Eintrag des Darstellungsvektors übersteigt.

(e) **durchschnittliche Eintragsgröße der Transformationsmatrix T ($\Delta_\infty(T)$)**
Desweiteren ist die durchschnittliche Eintragsgröße der Nicht–Nulleinträge der Transformationsmatrix deutlich über dem maximalen Eintrag des Darstellungsvektors, bleibt aber auch deutlich hinter dem maximalen Eintrag der Transformationsmatrix zurück. Die für den maximalen Eintrag des Darstellungsvektors festgestellten Abhängigkeiten übertragen sich wiederum eins zu eins auf die hier betrachtete Kenngröße.

(f) **Laufzeit (t)**
Die Laufzeiten bewegen sich zwischen 0.045 bis 4.077 CPU–Sekunden für $L_\infty(A) < 100$ bzw. 0.049 bis 8.3 CPU–Sekunden für $L_\infty(A) < 100000$. Somit ist auch diesmal die Laufzeit des Modulalgorithmus abhängig von der Größe des maximalen Eintrags der Startmatrix.

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2	1837	1	7.3	1.71731	4.077
450.1	90.02	99	1.1	1664.2	1.1	9.3	2.05684	3.402
403.9	80.78	99	1.1	1612.7	2.1	6.9	1.67006	3.325
350.7	70.14	99	1.1	1381.8	1.2	13.4	2.46468	2.602
295.6	59.12	98.9	1.1	1274.5	1.1	8.8	1.87321	2.216
253.4	50.68	98.8	1.2	1147.2	1.1	11.8	2.1053	1.893
198.7	39.74	98.7	1.5	1037.6	1.2	7.1	1.49431	1.523
156.3	31.26	98.6	1.4	889.5	1	18.4	2.33176	1.05
97	19.4	98.7	1.5	754.4	2.7	10.1	1.44724	0.676
51.6	10.32	98.1	1.9	625.4	4.2	13.9	1.42261	0.337
29.5	5.9	97.1	2	574.8	4.5	15	1.34951	0.225
11	2.2	93.6	2.5	525.8	7.5	34.5	1.38227	0.112
5.4	1.08	82.9	2	508.1	17.2	71.4	1.40366	0.074
2.8	0.56	62.7	2	503.8	11.5	37.6	1.16673	0.059
1.8	0.36	57.3	1.7	502	2.3	12.5	1.03446	0.054
0.5	0.1000000	18.8	1	500	1	1	1	0.045

Tabelle 10.38: $mgcd^{Pivot_4}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	4	2806.4	3.3	21	3.48347	5.764
449	89.8	997.6	1.7	2176.1	1.5	22.4	3.03373	5.476
398.2	79.64	996	1.7	2084.1	2.9	14	2.10479	4.854
348.9	69.78	996.6	1.7	1671.6	2.5	30.1	4.07921	3.465
303.2	60.64	996.2	2.1	1722.1	6	22	2.41089	3.57
243.8	48.76	995.1	2	1377.4	24.4	74.3	4.72804	2.721
197.8	39.56	992.8	1.8	1223.7	2	36.3	3.68857	2.086
149	29.8	992.9	2.1	1036.6	4.1	24.6	2.60081	1.458
98	19.6	987.6	2	836.3	25.1	80.4	4.41875	0.894
48.4	9.68	978.9	2.4	670.3	21.7	95.8	3.37207	0.452
25.7	5.14	962.9	2.9	585.1	16.3	118.5	2.98052	0.251
8.6	1.72	828.9	2.9	524.9	40.9	161.2	2.21147	0.116
6.2	1.24	887.6	2.5	512.1	104	491.8	4.86038	0.095
2.6	0.52	624.3	2.2	503.7	51	346.7	2.37066	0.066
2	0.4	567.1	1.7	502.3	65.2	391.3	3.10293	0.062
1.1	0.22	257.8	1.2	500.8	29.2	105.7	1.49141	0.049

Tabelle 10.39: $mgcd^{Pivot_4}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	6.2	3291.2	116.2	546	44.4912	6.379
455.2	91.04	9985	2.8	2845.2	4.6	35.4	3.61251	6.882
397.3	79.46	9961.8	2.5	2496	6.4	40.1	3.85958	6.118
352.3	70.46	9977.6	2.4	2243.6	5.2	46.2	4.08125	5.27
301.7	60.34	9968.3	2.5	1934.4	11.3	44.3	4.18543	4.262
252.5	50.5	9945.4	2.6	1752.4	82.7	234.8	11.1324	3.702
197.8	39.56	9954.8	2.7	1502.6	19.8	63.1	4.05018	2.967
151.5	30.3	9926	2.5	1117.4	430.5	1165.4	40.1986	1.825
99.9	19.98	9907.7	2.8	958.3	62.1	207.5	8.50203	1.263
51.9	10.38	9677.3	3.1	726.1	247	859.6	26.6677	0.638
26.4	5.28	9739.2	3.6	614.6	227.2	653.7	10.817	0.346
12.8	2.56	9122.1	3.2	541.4	328.6	2057.6	26.2859	0.18
4.2	0.84	8115.2	2.4	506.9	1278.3	4836.6	22.9797	0.087
3.9	0.78	8317.4	2.6	507.2	1172.3	4295.8	27.3941	0.084
1.7	0.34	4758.2	1.6	502.8	342	1134.2	6.60541	0.056
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.055

Tabelle 10.40: $mgcd^{Pivot_4}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	4.6	2709.2	4617.1	16760.9	1579.36	8.3
452.1	90.42	99823.7	3.3	3159.2	37.2	175.2	11.7776	7.864
399.3	79.86	99763.5	3.2	3008.6	48.5	234	12.3321	7.254
352.9	70.58	99657.1	2.9	2491.6	57.5	4457.5	308.817	6.153
307.4	61.48	99466.2	3.1	2168.7	1587.9	5320.6	226.346	5.279
251.8	50.36	99431.8	3.9	2181.8	71.2	211.9	9.81405	4.653
203.4	40.68	99511.4	3.4	1739.3	695.4	1485.2	49.2155	3.592
150.7	30.14	99297.6	3.7	1419.8	21.5	103.2	6.59734	2.616
104.3	20.86	98793.9	4	1105.8	73.3	543.4	18.2684	1.665
54.8	10.96	97747.7	4	783.4	4456	10642.4	234.164	0.838
24.6	4.92	96337.4	3.4	597.6	3464.4	16316.7	313.683	0.373
11.5	2.3	92767.5	3.4	543.2	3553.9	24232.9	239.507	0.199
5.1	1.02	79355.4	2.3	509.1	12918.5	43541.7	267.486	0.107
2.3	0.46	58436	1.8	503.1	8009.8	33739.7	172.252	0.071
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.053
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.049

Tabelle 10.41: $mgcd^{Pivot_4}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

Wir haben somit für die HNF–Berechnung vier einstufige Kriterien, nach denen wir das Pivotelement auswählen können. Bei allen wissen wir nicht, was beim Auftreten mehrerer Elemente, die die Pivoteigenschaft erfüllen, geschehen soll. Was liegt also näher, als die verschiedenen einstufigen Pivoteigenschaften zu kombinieren [HS79].

Wir erhalten dadurch folgende Kombinationskriterien:

	$Pivot_1$	$Pivot_2$	$Pivot_3$	$Pivot_4$
$Pivot_1$	–	$Pivot_{1+2}$	(nicht sinnvoll)	$Pivot_{1+4}$
$Pivot_2$	$Pivot_{2+1}$	–	$Pivot_{2+3}$	$Pivot_{2+4}$
$Pivot_3$	(nicht sinnvoll)	$Pivot_{3+2}$	–	$Pivot_{3+4}$
$Pivot_4$	$Pivot_{4+1}$	$Pivot_{4+2}$	$Pivot_{4+3}$	–

10.26. Bemerkung
Die Kombination des Pivotkriteriums $Pivot_1$ mit dem Pivotkriterium $Pivot_3$ ist offensichtlich nicht sinnvoll, weil die Auswahl des zweitgrößten Elements aus einer Menge gleicher Elemente keinen Sinn ergibt.

Für alle in der obigen Tabelle zusammengestellten zweistufigen Pivotkriterien wurden ebenfalls die zu Beginn dieses Abschnitts spezifizierten Testläufe durchgeführt. Die tabellarischen Zusammenstellungen finden sich im Anhang. Auf eine detaillierte Beschreibung der Eigenschaften verzichten wir an dieser Stelle zugunsten einer allgemeineren Feststellung.

Das erste Pivotkriterium dominiert das Verhalten des Algorithmus in Bezug auf die Größe und Anzahl der Elemente in der Transformationsmatrix und der Darstellung des *gcd* sowie in Bezug auf die Laufzeit. Das zweite Pivotkriterium übt lediglich einen *korrigierenden* Einfluß aus, da es eine Art Vorausschau in die Auswahl des Pivotelementes einbringt.

10.27. Bemerkung
Das Pivotkriterien $Pivot_{1+2}$ wurde bereits in der Arbeit [HS79] von Havas und Sterling untersucht und hat seine Leistungsfähigkeit unter Beweis gestellt.

10.1.1.3 mgcd–Algorithmen mit Konditionierung

Arne Storjohann [Sto96b, Sto97] geht in seinen Algorithmen zur Berechnung des mgcd und somit auch zur Berechnung der HNF einen anderen Weg. Die Idee seiner Algorithmen ist, die Berechnung des größten gemeinsamen Teilers eines Vektors ganzer Zahlen durch die Einführung eines Preconditioners auf die Berechnung des größten gemeinsamen Teilers zweier Zahlen zu reduzieren. Dazu gibt er in seiner Arbeit [Sto97] Algorithmen an, die für einen Vektor $v = (v_0, \ldots, v_{n-1}) \in \mathbb{Z}^n$ und eine ganze Zahl N einen Vektor $c = (c_0, \ldots, c_{n-1}) \in \mathbb{Z}^n$ berechnen, so daß gilt:

$$\begin{aligned} \gcd(\sum_{i=0}^{n-1} c_i \cdot v_i, N) &= \gcd(v_0, \ldots, v_{n-1}, N) \\ c_0 &= 1 \\ |c_i| &\leq \lceil 2 \cdot (\log_2(N)^{\frac{3}{2}} \rceil \qquad \forall 1 \leq i \leq (n-1). \end{aligned}$$

Desweiteren besitzt dieser Vektor c zusätzlich die Eigenschaft, daß höchstens $\lfloor \log_2(N) \rfloor$ viele Vektorelemente ungleiche dem Nullelement sind.

10.28. Bemerkung
Insbesondere bedeutet die obige Eigenschaft, daß die Bitlänge eines Vektorelementes v_i durch $O(log_2(log_2(N)))$ beschränkt ist. Andere Verfahren liefern Lösungsvektoren mit Vektorelementen in der Größenordnung $O(\log_2(N))$.

10.29. Bemerkung
Storjohann benennt diese spezielle Problemstellung als *Modulo N extended gcd problem.*

Es ist leicht zu sehen, daß die Algorithmen vor Storjohann sich leicht in der Art und Weise erweitern lassen, daß sie zu einem gegebenen Eingabevektor $v = (v_0, \ldots, v_{n-1}) \in \mathbb{Z}^n$ eine unimodulare Matrix $T \in \mathrm{GL}_n(\mathbb{Z})$ berechnen, so daß mit $w = T \cdot v$ gilt:

$$\gcd(w) \quad = \quad \gcd(w_0, w_1) = \gcd(v)$$

Desweiteren hat die Matrix T das Aussehen

$$T \quad = \quad \begin{pmatrix} 1 & 0 & 0 & 0 & \ldots & 0 \\ 0 & 1 & c_0 & c_1 & \ldots & c_{n-3} \\ 0 & 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 0 & 1 & \vdots & 0 \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 \end{pmatrix}$$

Zur praktischen Berechnung der Koeffizienten $c_0, \ldots, c_{n-3}$ gibt Storjohann in seiner Arbeit [Sto97] drei verschiedene Algorithmen bekannt. In LiDIA findet bisher lediglich der dort vorgestellte *brute force*–Algorithmus Verwendung, da er im Vergleich zu den beiden anderen Methoden am effizientesten arbeitet. Aus diesem Grund geben wir an dieser Stelle lediglich einen Konditionierungsalgorithmus an, der sich der "brute force" Methode bedient.

Der Algorithmus hat dann folgendes Aussehen:

10.30. Algorithmus

Konditionierungsalgorithmus: *cond_matrix*

EINGABE: $v \in \mathbb{Z}^n$.
AUSGABE: $T \in \mathrm{GL}_n(\mathbb{Z})$ mit $w = T \cdot v, \gcd(w_0, w_1) = \gcd(v)$.

[Initialisierung]
(1) $T = I_n$;
(2) $N = v_0$;
(3) $t = 0$;

```
(4)   g = 0;
(5)   a = v_1;
(6)   for (i = 2; i < n ∧ g ≠ 1; i++) do
(7)     if (v_i ≠ 0) then
(8)       if (a ≠ 0) then
(9)         g = gcd(v_i, a);
(10)        div_rem(qa, ra, a/g, N);
(11)        div_rem(qb, rb, v_i/g, N);
      [Konditionierung]
(12)        for (t = 0; gcd((ra + t * rb), N) ≠ 1; t++) do
(13)        od
(14)        a = a + t * v_i;
(15)        t_{1,i} = t;
(16)      else
(17)        a = a + v_i;
(18)        t_{1,i} = 1;
(19)      fi
(20)    fi
(21)  od
(22)  return (T);
```

Auf der Basis des Konditionierungsalgorithmus und den besonderen Eigenschaften der Konditionierungsmatrix läßt sich nun leicht ein mgcd–Algorithmus konstruieren, der sich als Modulalgorithmus eignet.

10.31. Algorithmus

mgcd–Algorithmus nach Storjohann: $mgcd_{Storjohann}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$
AUSGABE: $T \in \mathrm{GL}_n(\mathbb{Z})$ mit $a_{m-1,*} \cdot T = (0, \ldots, 0, \gcd(a_{m-1,*}))$.

```
(1)   v = a_{m-1,*};
      [Reduktion]
(2)   for (i = 1; i < n; i++;) do
(3)     pos_div_rem(q, r, v_i, v_0);
(4)     v_i = v_i − q · v_0;
(5)     t_{*,i} = t_{*,i} − q · t_{*,0};
(6)   od
      [Konditionierung]
(7)   T = cond_matrix(v);
(8)   v = v · T^T;
(9)   g = xgcd(v_0,v_1, a, b);
```

(10) $\mathrm{U} = \begin{pmatrix} a & -\frac{v_{n-1}}{g} & \\ b & \frac{v_{i-2}}{g} & \\ & & I_{n-2} \end{pmatrix}$;
(11) $T = T^T \cdot U$;
(12) $v = v \cdot U$;
[Elimination]
(13) **for** (i = 2; i < n; i++) **do**
(14) $v_i = t_{*,i} - \frac{v_i}{g} \cdot t_{*,0}$;
(15) **od**
(16) **return** (T);

Ergebnisse:
In den Tabellen 10.42, 10.43, 10.44 und 10.45 haben wir die Ergebnisse zusammengestellt, die der Algorithmus von Storjohann, angewendet auf unser Testszenario, liefert.

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	0.046
Bereich (bis)	2.9	1947.1	27889.9	0.2e10	0.6e9	1.922
abhängig von $L_0(v)$	$\surd$	$\surd$	$\surd$	–	–	$\surd$
abhängig von $L_\infty(A)$	–	–	$\surd$	$\surd\surd$	$\surd\surd$	$\surd$
abhängig von Eintragsverteilung von A	–	–	–	–	–	–

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2.3	1636.7	23.2	2296.8	499.758	1.878
450.1	90.02	99	2.4	1567.3	19.1	1890.9	469.472	1.697
403.9	80.78	99	2.6	1537.7	21.2	2098.8	546.859	1.527
350.7	70.14	99	2.3	1295.8	11.1	1098.9	252.7	1.337
295.6	59.12	98.9	2.4	1200.7	20.1	1989	461.963	1.141
253.4	50.68	98.8	2.5	1121.5	23	2275.7	329.104	0.976
198.7	39.74	98.7	2.3	951.6	24.8	2442.9	461.376	0.775
156.3	31.26	98.6	2.2	839.5	15.1	1488.7	210.127	0.613
97	19.4	98.7	2.4	729.8	20	1968.5	190.904	0.398
51.6	10.32	98.1	2.6	631	23.2	2264.8	179.927	0.228
29.5	5.9	97.1	2.4	566.2	27.2	2608	98.6653	0.146
11	2.2	93.6	2.4	524.5	15.7	1491.1	22.1407	0.085
5.4	1.08	82.9	2.5	510.8	23.3	1719.2	17.1482	0.06
2.8	0.56	62.7	2	504.1	15.5	672.3	6.91728	0.051
1.8	0.36	57.3	1.8	502.4	6.8	341.9	1.98627	0.05
0.5	0.1000000	18.8	1	500	1	1	1	0.046

Tabelle 10.42: $mgcd_{Storjohann}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.6	1796.3	436.1	435238	109446	1.908
449	89.8	997.6	2.5	1619.3	211.5	211161	58020	1.719
398.2	79.64	996	2.6	1534	223.4	222300	57029.6	1.535
348.9	69.78	996.6	2.3	1300.4	119.3	119002	28791.6	1.351
303.2	60.64	996.2	2.4	1225.4	324.7	322717	79039	1.178
243.8	48.76	995.1	2.4	1079.9	295.2	294270	56033.3	0.954
197.8	39.56	992.8	2.3	952.7	158.1	157115	29423.6	0.779
149	29.8	992.9	2.6	885	296.1	293830	51173.6	0.595
98	19.6	987.6	2.4	732.3	255.6	252737	30693.6	0.4
48.4	9.68	978.9	2.3	609	111.7	109371	7359.83	0.218
25.7	5.14	962.9	2.4	558.4	229.3	218012	8926.41	0.135
8.6	1.72	828.9	2.7	521	180	129704	1452.51	0.072
6.2	1.24	887.6	2.5	513.5	139.7	94663.1	965.534	0.063
2.6	0.52	624.3	2.1	503.9	62.4	7482.6	32.4102	0.051
2	0.4	567.1	1.7	502.5	85.4	33159.6	293.136	0.05
1.1	0.22	257.8	1.3	501.1	67.9	59275.3	341.748	0.048

Tabelle 10.43: $mgcd_{Storjohann}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2.7	1847.3	2056.2	0.205e8	0.469e7	1.927
455.2	91.04	9985	2.8	1772.3	1729.3	0.173e8	0.414e7	1.753
397.3	79.46	9961.8	2.6	1527.5	2589.3	0.258e8	0.673e7	1.537
352.3	70.46	9977.6	2.2	1274	3520.2	0.351e8	0.706e7	1.366
301.7	60.34	9968.3	2.4	1221.4	2131.8	0.213e8	0.484e7	1.173
252.5	50.5	9945.4	2.6	1158.7	3467.8	0.345e8	0.864e7	0.99
197.8	39.56	9954.8	2.8	1049.5	2505.3	0.249e8	0.375e7	0.784
151.5	30.3	9926	2.5	877.3	1248.4	0.124e8	0.201e7	0.604
99.9	19.98	9907.7	2.9	785	1821.7	0.181e8	0.199e7	0.41
51.9	10.38	9677.3	2.8	643.4	1958	0.192e8	0.143e7	0.231
26.4	5.28	9739.2	2.2	555.6	2077.2	0.204e8	745650	0.138
12.8	2.56	9122.1	2.4	528.2	1600.6	0.148e8	229910	0.086
4.2	0.84	8115.2	2.5	508.2	2330.6	0.137e8	110591	0.057
3.9	0.78	8317.4	2.2	506.8	2598.4	0.165e8	116508	0.056
1.7	0.34	4758.2	1.5	502.3	1400.3	0.811e7	49512	0.046
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.046

Tabelle 10.44: $mgcd_{Storjohann}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.9	1947.1	22634	0.226e10	0.635e9	1.922
452.1	90.42	99823.7	2.5	1626.5	14913.9	0.149e10	0.386e9	1.741
399.3	79.86	99763.5	2.5	1495.6	20497.8	0.205e10	0.535e9	1.542
352.9	70.58	99657.1	2.4	1339.4	24377.2	0.243e10	0.584e9	1.396
307.4	61.48	99466.2	2.5	1261.8	24617	0.245e10	0.484e9	1.202
251.8	50.36	99431.8	2.2	1051.4	24390.7	0.243e10	0.457e9	0.984
203.4	40.68	99511.4	2.1	926.2	19582.1	0.195e10	0.337e9	0.802
150.7	30.14	99297.6	2.7	905.8	26482.3	0.263e10	0.339e9	0.605
104.3	20.86	98793.9	2.6	764.4	15030.1	0.148e10	0.197e9	0.424
54.8	10.96	97747.7	2.5	633.7	27889.9	0.273e10	0.207e9	0.243
24.6	4.92	96337.4	2.8	569.2	23246.2	0.226e10	0.906e8	0.131
11.5	2.3	92767.5	2	521	17939.8	0.165e10	0.269e8	0.082
5.1	1.02	79355.4	2.1	508.5	13128.5	0.839e9	0.631e7	0.06
2.3	0.46	58436	1.7	503	5079.5	0.213e9	0.107e7	0.048
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.043
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.048

Tabelle 10.45: $mgcd_{Storjohann}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

Wir stellen fest, daß sich dieser Algorithmus bzgl. der von uns gewählten Kenngrößen wie der Modulalgorithmus $mgcd_{linear}$ verhält. Lediglich die Laufzeit wurde verkürzt. Diese Beobachtung läßt sich leicht erklären. Aufgrund der zufälligen Wahl der Einträge der Startmatrix entspricht der größte gemeinsame Teiler von 2 oder 3 Elementen bereits dem größten gemeinsamen Teiler der gesamten Zeile. Dies bedeutet, daß die Kenngrößen des hier betrachteten Modulalgorithmus vordergründig nicht durch die Berechnung der Konditionierungsmatrix sondern durch die Eliminationsphase dominiert werden. Da die Eliminationsphase des Modulalgorithmus $mgcd_{Storjohann}$ (wenn der größte gemeinsame Teiler bekannt ist) der (impliziten) Eliminationsphase des Modulalgorithmus $mgcd_{linear}$ entspricht, entsprechen sich auch in weiten Teilen die resultierenden Kenngrößen.

10.1.1.4 Hybrid–mgcd–Algorithmen

Ein anderer Ansatz, die mgcd-Berechnung zu verbessern, basiert auf der Beobachtung, daß der größte gemeinsame Teiler einer Menge ganzer Zahlen oft schon dem *gcd* von zwei bzw. drei dieser Zahlen entspricht. Leider führt uns diese Fragestellung direkt zu dem **MINIMUM–GCD–SET–Problem**, welches wie wir schon erwähnt haben, aufgrund eines Resultats aus [MH94b, MH94a] NP–vollständig ist.

Der folgende parametrisierte Algorithmus stellt eine Variante der bereits vorgestellten Algorithmen von Havas dar. In ihn wurde zusätzlich die Heuristik eingebracht, daß der *gcd* der Elemente einer Zeile mit n Elementen sehr oft gleich dem *gcd* der ersten beiden Elemente ist.

10.32. Algorithmus

mgcd–Algorithmus mit gcd–Heuristik: $mgcd_{Heuristik}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$
AUSGABE: $T \in \mathrm{GL}_n(\mathbb{Z})$ mit $a_{m-1,*} \cdot T = (0, \ldots, 0, \gcd(a_{m-1,*}))$.

(1) $T = I_n$;
[Phase I: Heuristik]
(2) g = xgcd(v_{i-1},v_i, a, b);
(3) $\mathrm{U} = \begin{pmatrix} I_{i-1} & & & \\ & -\frac{v_i}{g} & a & \\ & \frac{v_{i-1}}{g} & b & \\ & & & I_{n-i} \end{pmatrix}$;
(4) $v = v \cdot U$;
(5) $T = T \cdot U$;
[Phase II: Havas Algorithmus]
(6) $T' = mgcd_{BestRemainder}(A, v)$; // Havas mgcd–Algorithmus
(7) $T = T \cdot T'$;
(8) **return** (T);

10.33. Bemerkung
Diese Idee wird auch in den in Pari [BBCO99] integierten Algorithmen verwendet.

Ergebnisse:
Die Analyse dieses Hybrid–Verfahrens führt zu den in den Tabellen 10.46, 10.47, 10.48 und 10.49 zusammengestellten Ergebnissen.

	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
Bereich (von)	1	500	1	1	1	0.048
Bereich (bis)	2.8	1696.3	0.7e11	0.5e16	0.1e16	1.449
abhängig von $L_0(v)$	$\surd$	$\surd$	$\surd$	–	–	–
abhängig von $L_\infty(A)$	–	–	$\surd$	$\surd\surd$	$\surd\surd$	–
abhängig von Eintragsverteilung von A	–	–	–	–	–	–

Die Ergebnisse dieses Algorithmus entsprechen in weiten Teilen denen des Algorithmus, der in der zweiten Phase verwendet wird, d.h. in obigem Fall den Ergebnissen des Modulalgorithmus $mgcd_{BestRemainder}$. Die heuristische Erweiterung führt vorrangig zu einer kürzeren Laufzeit. Auf eine detaillierte Analyse der Ergebnisse können wir aus diesem Grund an dieser Stelle verzichten.

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	1	994.3	1	99	25.4873	1.415
450.1	90.02	99	1	944.6	1	99	23.6107	1.279
403.9	80.78	99	1	898.2	1	99	22.834	1.146
350.7	70.14	99	1	846.3	1	99	21.1815	1
295.6	59.12	98.9	1	791.8	1	98.9	18.9975	0.844
253.4	50.68	98.8	1.1	775.1	3.8	376	66.2518	0.726
198.7	39.74	98.7	1.4	774.4	15.6	1533.5	201.72	0.573
156.3	31.26	98.6	1.3	701.6	6.4	625	73.0442	0.455
97	19.4	98.7	1.3	624.1	7.1	701	64.5544	0.293
51.6	10.32	98.1	1.9	594.6	20.6	2018.1	93.1483	0.172
29.5	5.9	97.1	1.8	550.7	18.9	1822.4	52.3223	0.119
11	2.2	93.6	2.3	522.4	49.5	4734.6	51.0008	0.073
5.4	1.08	82.9	2	508	227.1	19876.7	115.304	0.056
2.8	0.56	62.7	2	503.9	23	840.7	4.55408	0.051
1.8	0.36	57.3	1.6	501.9	3.4	45.8	1.14784	0.05
0.5	0.1000000	18.8	1	500	1	1	1	0.048

Tabelle 10.46: $mgcd_{Heuristik}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	1.4	1197.8	102.2	101833	21259.1	1.439
449	89.8	997.6	1.6	1213.6	6684.6	0.668e7	0.128e7	1.296
398.2	79.64	996	1.9	1253.7	1909.4	0.189e7	300141	1.151
348.9	69.78	996.6	1.7	1086.8	129.1	128735	20064.6	1.015
303.2	60.64	996.2	1.8	1045.1	111.3	110933	16297.5	0.881
243.8	48.76	995.1	1.8	934.6	98.2	97830.7	12983.2	0.711
197.8	39.56	992.8	1.8	853.9	1185.4	0.117e7	133473	0.58
149	29.8	992.9	1.9	777.1	127.3	126352	12418.4	0.442
98	19.6	987.6	2	694.9	125	124217	10047.1	0.304
48.4	9.68	978.9	2.3	606.3	61112.8	0.609e8	0.218e7	0.169
25.7	5.14	962.9	2.1	551.3	522.2	490672	12861.5	0.11
8.6	1.72	828.9	2.1	516.2	457.9	369502	5885.68	0.067
6.2	1.24	887.6	2.2	510.9	6249.5	0.491e7	15147.9	0.059
2.6	0.52	624.3	2.3	504.1	3846	0.268e7	8589.04	0.051
2	0.4	567.1	1.6	502.2	100.6	49997.4	226.127	0.05
1.1	0.22	257.8	1.2	500.8	49.8	43340.9	209.062	0.05

Tabelle 10.47: $mgcd_{Heuristik}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2.4	1696.3	0.163375e8	0.163e12	0.253e11	1.449
455.2	91.04	9985	2.2	1498.2	464361	0.464e10	0.698e9	1.325
397.3	79.46	9961.8	2.1	1332.1	348176	0.347e10	0.523e9	1.166
352.3	70.46	9977.6	2.3	1306.7	460721	0.460e10	0.588e9	1.033
301.7	60.34	9968.3	2.5	1251.4	384730	0.384e10	0.479e9	0.892
252.5	50.5	9945.4	2.3	1073.7	824899	0.822e10	0.995e9	0.745
197.8	39.56	9954.8	2.4	970.9	231202	0.230e10	0.232e9	0.59
151.5	30.3	9926	2.2	829	586967	0.586e10	0.528e9	0.456
99.9	19.98	9907.7	2.4	735.8	0.931e7	0.922e11	0.641e10	0.312
51.9	10.38	9677.3	2.4	621.1	220744	0.214e10	0.905e8	0.18
26.4	5.28	9739.2	2.6	565.4	0.157e9	0.147e13	0.326e11	0.108
12.8	2.56	9122.1	2.5	528.7	411529	0.402e10	0.397e8	0.078
4.2	0.84	8115.2	2.4	507.1	428869	0.818e9	0.309e7	0.055
3.9	0.78	8317.4	2.2	505.9	550274	0.761e7	37621.6	0.054
1.7	0.34	4758.2	1.4	502	391.3	666711	4963.51	0.049
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.049

Tabelle 10.48: $mgcd_{Heuristik}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.2	1597.1	0.107e8	0.107e13	0.167e12	1.445
452.1	90.42	99823.7	2.5	1627.4	0.732e11	0.731e16	0.101e16	1.325
399.3	79.86	99763.5	2.5	1493.5	0.615e7	0.608e12	0.802e11	1.176
352.9	70.58	99657.1	2.1	1239.7	0.243e7	0.242e12	0.348e11	1.04
307.4	61.48	99466.2	2.5	1265.2	0.559e11	0.557e16	0.664e15	0.919
251.8	50.36	99431.8	2.2	1047	0.165e8	0.164e13	0.188e12	0.747
203.4	40.68	99511.4	2.8	1068	0.199e11	0.197e16	0.203e15	0.615
150.7	30.14	99297.6	2.5	873.3	0.222e8	0.220e13	0.182e12	0.459
104.3	20.86	98793.9	2.4	747.4	0.291e8	0.289e13	0.206e12	0.324
54.8	10.96	97747.7	2	607.6	16226.1	0.159e10	0.809e8	0.187
24.6	4.92	96337.4	2.4	558.3	0.105e10	0.101e15	0.202e13	0.107
11.5	2.3	92767.5	2.3	524.4	0.285e8	0.273e13	0.325e11	0.073
5.1	1.02	79355.4	2.4	510.1	0.279e8	0.207e13	0.152e11	0.058
2.3	0.46	58436	1.8	503.1	0.392e8	0.556e9	0.180e7	0.052
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.047
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.047

Tabelle 10.49: $mgcd_{Heuristik}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

10.34. Bemerkung
Diese heuristische Erweiterung ist wiederum exemplarisch zu verstehen. Prinzipiell kann man jeden der bisher vorgestellten Algorithmen um diese heuristische Komponente erweitern.

10.1.1.5 Entscheidungsmatrix *mgcd*–Algorithmen

Auf der Basis der Laufzeitergebnisse unseres Experimentes haben wir die folgende Entscheidungsmatrix erstellt.

Name	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)(\Delta_\infty(T))$	t
$mgcd_{linear}$	3	2	7	7	5
$mgcd_{Bradley}$	4	4	6	4	9
$mgcd_{Ilio}$	2	3	6	6	5
$mgcd_{opt}$	3	3	8	7	3
$mgcd_{Blankinship}$	3	2	7	7	5
$mgcd_{BestRemainder}$	1	1	5	5	2
$mgcd^{Pivot_2}_{BestRemainder}(k=1)$	5	5	3	3	7
$mgcd^{Pivot_2}_{BestRemainder}(k=2)$	4	5	2	2	8
$mgcd^{Pivot_3}_{BestRemainder}$	5	6	1	1	6
$mgcd^{Pivot_4}_{BestRemainder}$	4	4	4	4	6
$mgcd_{Storjohann}$	3	2	7	7	4
$mgcd_{Heuristik}$	1	1	5	5	1

In ihr stellen wir *mgcd*–Algorithmen in Beziehung zu den betrachteten Kenngrößen. Die Zahlenwerte geben die Rangfolge an, in welcher die *mgcd*–Algorithmen einer Minimierung der jeweiligen Kenngröße am besten gerecht werden.

10.1.2 Normalisierung

In diesem Abschnitt gehen wir auf den zweiten der beiden Modulalgorithmen ein und illustrieren verschiedene Methoden, die Nebendiagonalelemente einer oberer Dreiecksmatrix modulo dem zugehörigen Diagonalelement zu reduzieren.

Wir unterscheiden die folgenden sechs Methoden, welche sich in in drei Gruppen gliedern.

- **Einfache Normalisierungmethoden**
  Die erste Gruppe umfaßt die folgenden beiden Normalisierungsroutinen.

  1. **Standardnormalisierung**
     Die einfachste Methode, die wir als *Standardnormalisierung* bezeichnen, reduziert die Nebendiagonalelemente zeilenweise beginnend bei der letzten Zeile modulo dem entsprechenden Diagonalelement. Die Reihenfolge, in der die Nichtdiagonalelemente normalisiert werden, entspricht der in der folgenden Skizze angedeuteten Reihenfolge.

$$\begin{pmatrix} * & 4 & 5 & 6 \\ & * & 2 & 3 \\ & & * & 1 \\ & & & * \end{pmatrix}$$

Prinzip: von unten nach oben, von links nach rechts

Daraus ergibt sich der folgende Algorithmus.

10.35. Algorithmus

Standardnormalisierung: $Normalize_{Std}$

EINGABE: Obere Dreiecksmatrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, $\text{rank}_{\mathbb{Z}}(A) = m$,
Startzeile: *startr*, Startspalte: *startc*
AUSGABE: normalisierte, obere Dreiecksmatrix

```
    [Normalisierung]
(1)  for (i = n - 2, j = m - 2; i ≥ startc ∧ j ≥ startr; i– –, j– –) do
(2)    for (l = i + 1; l < n; l++) do
(3)      if (a_{j,l} ≠ 0) then
(4)        pos_div_rem(q, r, a_{j,l}, a_{j,i});
(5)        a_{*,l} = a_{*,l} − q · a_{*,i};
(6)      fi
(7)    od
(8)  od
(9)  return (A);
```

2. **Normalisierung nach Chou–Collins**
   Chou und Collins geben in ihrer Arbeit [CC82] eine alternative Methode an, die in der Praxis zu besseren Resultaten führt. Der Unterschied besteht in der Reihenfolge, in der die Nebendiagonalelemente modulo dem jeweiligen Diagonalelement normalisiert werden. Durch die Änderung der Reihenfolge werden in späteren Normalisierungsschritten bereits normalisierte Einträge verwendet, was zu kleineren Zwischeneinträgen führt.
   In der folgenden Skizze ist die modifizierte Reihenfolge angegeben.

$$\begin{pmatrix} * & 1 & 3 & 6 \\ & * & 2 & 5 \\ & & * & 4 \\ & & & * \end{pmatrix}$$

Prinzip: links nach rechts, von unten nach oben

Dies führt zu dem folgenden Algorithmus.

10.36. Algorithmus

Normalisierung nach Chou–Collins: $Normalize_{ChouCollins}$

EINGABE: Obere Dreiecksmatrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, $\text{rank}_{\mathbb{Z}}(A) = m$,
Startzeile: *startr*, Startspalte: *startc*
AUSGABE: normalisierte, obere Dreiecksmatrix

```
    [Normalisierung]
(1)   for (i = startc + 1, p = startr; i < n; i++, p++) do
(2)     for (k = i - 1, l = p; k ≥ startc; k– –, l– –) do
(3)       if (a_{l,i} ≠ 0∧ a_{l,k} ≠ 0) then
(4)         pos_div_rem(q, r, a_{l,i}, a_{l,k});
(5)         a_{*,i} = a_{*,i} − q · a_{*,k};
(6)       fi
(7)     od
(8)   od
(9)   return (A);
```

10.37. Bemerkung
Die beiden einfachen Normalisierungsmethoden lassen sich leicht in der Art und Weise modifizieren, daß sie auch Matrizen ohne vollen Zeilenrang normalisieren können.

- **Modulare Normalisierungsmethoden**
  Die zweite Gruppe von Normalisierungsroutinen macht sich die in Satz 2.28 bewiesene Eigenschaft zunutze, daß der Vektor $\alpha \cdot e_i$ ein Element des durch die Spalten einer Matrix A induzierten Gitters Ł(A) ist, wenn α ein ganzzahliges Vielfaches der Gitterdeterminanten von Ł(A) ist. Dem entsprechend setzen die Algorithmen dieser Gruppe vollen Zeilenrang zwingend voraus.

  In einer ersten Phase berechnen die Methoden dieser Gruppe zunächst α und nutzen diese Information im weiteren Normalisierungsprozeß zur Kontrolle der Größe

der Zwischenergebnisse. Es ist leicht zu sehen, daß die Korrektheit des Ergebnisses durch die Reduktion nicht beeinflußt wird. Somit ergeben sich die folgenden beiden Normalisierungroutinen.

1. **Modulare Standardnormalisierung**
   Als Normalisierungsreihenfolge wird die Reihenfolge der Standardnormalisierung verwendet, d.h.

$$\begin{pmatrix} * & 4 & 5 & 6 \\ & * & 2 & 3 \\ & & * & 1 \\ & & & * \end{pmatrix}$$

   Prinzip: von unten nach oben, von links nach rechts

   Daraus ergibt sich der folgende Algorithmus.

10.38. Algorithmus

Modulare Standardnormalisierung: $NormalizeMod_{Std}$

EINGABE: Obere Dreiecksmatrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, $\mathrm{rank}_{\mathbb{Z}}(A) = m$,
Startzeile: *startr*, Startspalte: *startc*
AUSGABE: normalisierte, obere Dreiecksmatrix

```
     [Initialisierung]
(1)    α = 1;
     [Berechnung der Gitterdeterminanten]
(2)    for (i = startr, j = startc; i < n ∧ j < m; i++, j++) do
(3)      α = α · |a_{i,j}|;
(4)    od
     [Reduktion]
(5)    A = A mod (2 · α);
     [Normalisierung]
(6)    for (i = n - 1, j = m - 1; i ≥ startc ∧ j ≥ startr; i– –, j– –) do
(7)      for (l = i + 1; l < n; l++) do
(8)        if (a_{j,l} ≠ 0) then
(9)          pos_div_rem(q, r, a_{j,l}, a_{j,i});
(10)         a_{*,l} = (a_{*,l} − q · a_{*,i}) mod (2 · α);
(11)       fi
(12)     od
(13)   od
(14)   return (A);
```

10.39. Bemerkung [LiDIA]
Der Operator $r = a \bmod b$ reduziert a modulo b und liefert den betragsmäßig kleinsten Rest zurück.

2. **Modulare Normalisierung nach Chou–Collins**
   Kombiniert man die Idee der Verwendung der Modulreduktion mit der Norma-

lisierungsreihenfolge von Chou–Collins, so erhält man den folgenden Algorithmus.

$$\begin{pmatrix} * & 1 & 3 & 6 \\ & * & 2 & 5 \\ & & * & 4 \\ & & & * \end{pmatrix}$$

Prinzip: links nach rechts, von unten nach oben

Dies führt zu dem folgenden Algorithmus.

10.40. Algorithmus

Modulare Normalisierung nach Chou–Collins:
$NormalizeMod_{ChouCollins}$

EINGABE: Obere Dreiecksmatrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, $\mathrm{rank}_{\mathbb{Z}}(A) = m$,
Startzeile: *startr*, Startspalte: *startc*
AUSGABE: normalisierte, obere Dreiecksmatrix

```
    [Initialisierung]
(1)   α = 1;
    [Berechnung der Gitterdeterminanten]
(2)   for (i = startr, j = startc; i < n ∧ j < m; i++, j++) do
(3)     α = α · |a_{i,j}|;
(4)   od
    [Reduktion]
(5)   A = A mod (2 · alpha);
    [Normalisierung]
(6)   for (i = startc + 1, p = startr; i < n; i++, p++) do
(7)     for (k = i - 1, l = p; k ≥ startc; k– –, l– –) do
(8)       if (a_{l,i} ≠ 0∧ a_{l,k} ≠ 0) then
(9)         pos_div_rem(q, r, a_{l,i}, a_{l,k});
(10)        a_{*,i} = (a_{*,i} − q · a_{*,k}) mod (2 · α);
(11)      fi
(12)    od
(13)  od
(14)  return (A);
```

Dieser Algorithmus wurde bereits in der Arbeit [Sto96a] vorgestellt, in der auch der folgende Satz bewiesen wurde.

10.41. Satz
Der Algorithmus $NormalizeMod_{ChouCollins}$ *ist ein deterministischer Algorithmus, der als Eingabe eine ganzzahlige obere Dreiecksmatrix* $A \in \mathbf{Mat}_{n\times n}(\mathbb{Z})$ *mit* $\mathrm{rank}_{\mathbb{Z}}(A) = n$ *erhält und als Ausgabe die HNF von A zurückgibt. Wenn sowohl* $|\det_{\mathbb{Z}}(A)|$ *als auch* $\|A\|$ *durch D beschränkt sind, ist die Bit–Komplexität dieses Algorithmus durch* $O(n^2 \cdot \log_2^2(D))$ *beschränkt.*

Beweis: Siehe [Sto96a] ■

- **Hybrid–Normalisierungsmethoden**
  Die dritte Gruppe umfaßt Normalisierungsmethoden, die speziell für die HNF–Berechnung im Umfeld der Klassengruppenberechnung [Jr.99, Nei01] entwickelt wurden. Sie basieren auf der Beobachtung, daß in vielen Fällen über 90% der Diagonaleinträge der HNF aus Einheitseinträgen (± 1) bestehen. Dies führt dazu, daß alle Nebendiagonalelemente der jeweiligen Zeilen in der Normalisierungsphase durch eine geeignete Addition eines Vielfachen der aktuellen Arbeitsspalte (Diagonalposition) eliminiert werden und somit durch die Reduktion der Eintragsdichte der Restmatrix die Ausführungseffizienz folgender Operationen erhöhen [Sto96a].

  Aus diesem Grund ist die folgende in zwei Phasen eingeteilte Vorgehensweise sinnvoll:

  1. In der ersten Phase gehen wir zeilenweise von oben nach unten vor und reduzieren die Nebendiagonalelemente der Zeilen, die eine Einheit als Diagonalelement besitzen.
  2. In der zweiten Phase reduzieren wir die bisher nicht reduzierten Nebendiagonalelemente modulo dem zugehörigen Diagonalelement. Dazu bieten sich prinzipiell zum einem die einfachen und zum anderen die modularen Normalisierungsmethoden an, die wir bereits vorgestellt haben.

  Aufgrund der praktischen Überlegung, die Anwendbarkeit der Normalisierungsalgorithmen dieser Gruppe nicht nur auf Matrizen mit vollem Zeilenrang einzuschränken, werden in LiDIA für die zweite Phase lediglich die einfachen Normalisierungsmethoden verwendet, d.h. es ergeben sich die folgenden beiden Normalisierungsroutinen:

  1. **Hybrid–Standardnormalisierung (2-Phasen–Normalisierung nach Jacobson)**

10.42. Algorithmus

Hybrid–Standardnormalisierung:
$NormalizeHybrid_{Std}$

EINGABE: Obere Dreiecksmatrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, $\mathrm{rank}_{\mathbb{Z}}(A) = m$,
Startzeile: *startr*, Startspalte: *startc*
AUSGABE: normalisierte, obere Dreiecksmatrix

```
    [Phase 1]
(1)   for (i = startr, j = startc; i < m ∧ j < n; i++, j++) do
(2)     if (|a_{i,j}| = 1) then
(3)       for (l = j + 1; l < n; l++) do
(4)         if (a_{i,l} ≠ 0) then
(5)           a_{*,l} = a_{*,l} − a_{i,l} · a_{*,j};
(6)         fi
(7)       od
(8)     fi
(9)   od
    [Phase 2]
(10)  Normalize_{Std}(A, startr, startc);
```

```
(11)   return (A);
```

2. **Hybrid–Normalisierung nach Chou–Collins**

10.43. Algorithmus

Hybrid–Normalisierung:
$NormalizeHybrid_{ChouCollins}$

EINGABE: Obere Dreiecksmatrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, $\mathrm{rank}_{\mathbb{Z}}(A) = m$,
Startzeile: *startr*, Startspalte: *startc*
AUSGABE: normalisierte, obere Dreiecksmatrix

```
       [Phase 1]
(1)    for (i = startr, j = startc; i < m ∧ j < n; i++, j++) do
(2)      if (|a_{i,j}| = 1) then
(3)        for (l = j + 1; l < n; l++) do
(4)          if (a_{i,l} ≠ 0) then
(5)            a_{*,l} = a_{*,l} − a_{i,l} · a_{*,j};
(6)          fi
(7)        od
(8)      fi
(9)    od
       [Phase 2]
(10)   Normalize_{ChouCollins}(A, startr, startc);
(11)   return (A);
```

Praktische Ergebnisse

Im folgenden untersuchen wir das Verhalten der vorgestellten Normalisierungsverfahren.

1. Dazu wenden wir in einer ersten Testreihe die einzelnen Normalisierungsverfahren auf obere Dreiecksmatrizen steigender Dimension mit Diagonaleinträgen im Intervall $[-100; 100]$ und Nebendiagonaleinträgen im Intervall $[-10^5; 10^5]$ an und protokollieren den dabei maximal entstehenden Zwischeneintrag sowie die zur Normalisierung benötigte Rechenzeit. Als Matrixrepräsentation wird die zeilenorientierte, dichtbesetzte Repräsentation verwendet.

   Berechnungen, die länger als eine Stunde benötigen (++) bzw. mehr als 100 MB Hauptspeicher verbrauchen (– –), werden vorzeitig beendet.

   Die über 10 Durchführungen gemittelten Ergebnisse haben wir in den Tabellen 10.50, 10.51, 10.52, 10.53, 10.54 und 10.55 zusammengefaßt.

   Es ist deutlich zu erkennen, daß Normalisierungen, die sich der Normalisierungsreihenfolge nach Chou–Collins bedienen, Einträge geringerer Größe im Vergleich zu den jeweiligen Standardnormalisierungen errechnen und somit eine geringere

Dimension	Laufzeit	maximaler Eintrag
100×100	83 hsec	$> 10^{295}$
200×200	9 sec 3 hsec	$> 10^{600}$
300×300	35 sec 86 hsec	$> 10^{901}$
400×400	1 min 36 sec 89 hsec	$> 10^{1206}$
500×500	3 min 30 sec 30 hsec	$> 10^{1489}$
600×600	6 min 47 sec 37 hsec	$> 10^{1800}$
700×700	– –	– –
800×800	– –	– –
900×900	– –	– –
1000×1000	– –	– –

Tabelle 10.50: Standardnormalisierung: Laufzeit, maximaler Eintrag

Dimension	Laufzeit	maximaler Eintrag
100×100	57 hsec	37042744
200×200	5 sec 20 hsec	39735024398
300×300	18 sec 6 hsec	2194890782175
400×400	43 sec 20 hsec	15496208142734
500×500	1 min 24 sec 93 hsec	3296382063803804
600×600	2 min 28 sec 52 hsec	3545766461146196267
700×700	3 min 56 sec 99 hsec	84953292921999057957
800×800	5 min 56 sec 95 hsec	23018441862517240462109
900×900	8 min 32 sec 18 hsec	10533309070434945828029109
1000×1000	12 min 5 sec 81 hsec	– –

Tabelle 10.51: Normalisierung nach Chou–Collins: Laufzeit, maximaler Eintrag

Dimension	Laufzeit
100×100	4 sec 2 hsec
200×200	44 sec 46 hsec
300×300	3 min 11 sec 46 hsec
400×400	9 min 30 sec 77 hsec
500×500	21 min 25 sec 61 hsec
600×600	42 min 23 sec 98 hsec
700×700	++
800×800	++
900×900	++
1000×1000	++

Tabelle 10.52: Modulare Standardnormalisierung: Laufzeit

Dimension	Laufzeit
100×100	3 sec 27 hsec
200×200	36 sec 58 hsec
300×300	2 min 33 sec 67 hsec
400×400	7 min 26 sec 53 hsec
500×500	17 min 7 sec 4 hsec
600×600	34 min 43 sec 20 hsec
700×700	++
800×800	++
900×900	++
1000×1000	++

Tabelle 10.53: Modulare Normalisierung nach Chou–Collins: Laufzeit

Dimension	Laufzeit	maximaler Eintrag
100×100	83 hsec	$> 10^{295}$
200×200	9 sec 1 hsec	$> 10^{600}$
300×300	36 sec 41 hsec	$> 10^{901}$
400×400	1 min 37 sec 82 hsec	$> 10^{1206}$
500×500	3 min 35 sec 85 hsec	$> 10^{1487}$
600×600	6 min 51 sec 76 hsec	$> 10^{1800}$
700×700	– –	– –
800×800	– –	– –
900×900	– –	– –
1000×1000	– –	– –

Tabelle 10.54: Hybrid–Standardnormalisierung: Laufzeit, maximaler Eintrag

Dimension	Laufzeit	maximaler Eintrag
100×100	58 hsec	375887623320
200×200	5 sec 25 hsec	2117760075113803302332052
300×300	18 sec 50 hsec	357124062777769923260086312725035
400×400	43 sec 54 hsec	6930185381072158002
500×500	1 min 25 sec 63 hsec	422290402353359657169697
600×600	2 min 29 sec 51 hsec	3700002386522142702919041
700×700	4 min 45 hsec	21263670911180440697072161069356
800×800	5 min 58 sec 27 hsec	23014359688426356766605
900×900	8 min 42 sec 59 hsec	$> 5099 \cdot 10^{50}$
1000×1000	11 min 52 sec 69 hsec	5110855972402412741992123167020905

Tabelle 10.55: Hybrid–Normalisierung nach Chou–Collins: Laufzeit, maximaler Eintrag

Ausführungszeit benötigen. Desweiteren fällt auf, daß die Verwendung einer modularen Vorgehensweise auf den ersten Blick keine Vorteile gegenüber nichtmodularen Vorgehensweisen erkennen läßt. Auf den zweiten Blick stellen wir aber fest, daß die Berechnungen nicht aus Mangel an Hauptspeicherplatz, sondern wegen Überschreitung der vorgegebenen Rechenzeitschranke beendet wurden. Die zusätzlichen Moduloreduktionen verursachen den erhöhten Laufzeitbedarf.

2. In einer zweiten Testreihe untersuchen wir den Einfluß des prozentualen Anteils an Einheitseinträgen auf der Diagonalen auf die Gesamtlaufzeit der Normalisierung. Zu diesem Zweck fixieren wir eine (500×500)–Matrix mit Nebendiagonaleinträgen im Intervall $[-10^5; 10^5]$. Die Diagonale besetzen wir mit Einträgen aus dem Intervall $[-100; 100]$, wobei wir den Anteil der Einheiten (± 1) von 50% bis 99% skalieren. Als Matrixrepräsentation wird wiederum die zeilenorientierte, dichtbesetzte Repräsentation verwendet.

   Die über 10 Durchführungen gemittelten Ergebnisse sind in den Tabellen 10.56, 10.57 und 10.58 dargestellt.

Prozent	Laufzeit	
	Standard	*ChouCollins*
50%	4 min 3 sec 29 hsec	1 min 22 sec
60%	4 min 6 sec 18 hsec	1 min 21 sec 20 hsec
70%	4 min 12 sec 72 hsec	1 min 20 sec 69 hsec
80%	4 min 17 sec 83 hsec	1 min 18 sec 72 hsec
90%	4 min 20 sec 47 hsec	1 min 17 sec 85 hsec
95%	4 min 25 sec 14 hsec	1 min 16 sec 72 hsec
96%	4 min 28 sec 14 hsec	1 min 16 sec 51 hsec
97%	4 min 26 sec 46 hsec	1 min 16 sec 53 hsec
98%	4 min 27 sec 36 hsec	1 min 16 sec 42 hsec
99%	4 min 27 sec 51 hsec	1 min 16 sec 19 hsec

Tabelle 10.56: Einfache Normalisierungsmethoden: Einheitenabhängigkeit

Prozent	Laufzeit	
	Standard	*ChouCollins*
50%	12 min 59 sec 94 hsec	8 min 23 sec 57 hsec
60%	11 min 38 sec 78 hsec	7 min 25 sec 40 hsec
70%	10 min 7 sec 15 hsec	6 min 17 sec 13 hsec
80%	8 min 10 sec 63 hsec	4 min 48 sec 2 hsec
90%	7 min 10 sec 65 hsec	4 min 10 sec 3 hsec
95%	5 min 17 sec 64 hsec	2 min 59 sec 37 hsec
96%	5 min 11 sec 34 hsec	2 min 56 sec 93 hsec
97%	4 min 53 sec 60 hsec	2 min 51 sec 53 hsec
98%	4 min 41 sec 56 hsec	2 min 43 sec 74 hsec
99%	4 min 9 sec 75 hsec	2 min 41 sec 96 hsec

Tabelle 10.57: Modulare Normalisierungsmethoden: Einheitenabhängigkeit

Prozent	Laufzeit	
	Standard	*ChouCollins*
50%	5 min 53 sec 33 hsec	2 min 10 sec 56 hsec
60%	5 min 54 hsec	2 min 5 sec 46 hsec
70%	4 min 24 sec 57 hsec	2 min 8 sec 73 hsec
80%	2 min 46 sec 90 hsec	1 min 52 sec 64 hsec
90%	2 min 8 sec 4 hsec	1 min 51 sec 80 hsec
95%	1 min 31 sec 54 hsec	1 min 37 sec 91 hsec
96%	1 min 26 sec 86 hsec	1 min 25 sec 27 hsec
97%	1 min 28 sec 38 hsec	1 min 25 sec 28 hsec
98%	1 min 23 sec 71 hsec	1 min 21 sec 85 hsec
99%	1 min 21 sec 5 hsec	1 min 19 sec 51 hsec

Tabelle 10.58: Hybrid–Normalisierungsmethoden: Einheitenabhängigkeit

Die einfachen Normalisierungsmethoden zeigen keine Abhängigkeit von der Anzahl der Einheiten auf der Diagonalen im Gegensatz zu den modularen und den Hybrid–Normalisierungsmethoden. Die Abhängigkeit der modularen Normalisierungsmethoden ist auf die kleinere Gitterdeterminante der betrachteten Matrix zurückzuführen und somit lediglich ein Nebeneffekt, währenddessen die Abhängigkeit der Hybrid–Methoden ein beabsichtigte Algorithmuseigenschaft ist.

10.1.3 Algorithmenübersicht

In den vergangenen Abschnitten haben wir, die Kernalgorithmen der zeilenweisen HNF–Berechnung, die Modulalgorithmen zur Normalisierung und die Modulalgorithmen zur mgcd–Berechnung vorgestellt. In diesem Abschnitt widmen wir uns der Aufgabe zu illustrieren, welche Algorithmen sich aus der Kombination von Kern– mit Modulalgorithmen ergeben.

In der aktuellen LiDIA Implementierung sind gemäß Tabelle 10.59 6 Normalisierungsalgorithmen implementiert.

Kennzahl	**Normalisierung**	**Bezeichnung**
0	$Normalize_{Std}$	Standardnormalisierung
1	$Normalize_{ChouCollins}$	Normalisierung nach Chou–Collins
2	$NormalizeMod_{Std}$	Modulare Standardnormalisierung
3	$NormalizeMod_{ChouCollins}$	Modulare Normalisierung nach Chou–Collins
4	$NormalizeHybrid_{Std}$	Hybrid–Standardnormalisierung
5	$NormalizeHybrid_{ChouCollins}$	Hybrid–Normalisierung nach Chou–Collins

Tabelle 10.59: Zusammenstellung normalize–Algorithmen

Dazu kommen 22 *mgcd*–Modulalgorithmen (Tabelle 10.60).

Kennzahl	mgcd–Algorithmus	Bezeichnung
0	$mgcd_{linear}$	Linear, iterativer Algorithmus
1	$mgcd_{Bradley}$	Algorithmus von Bradley
2	$mgcd_{Ilio}$	Algorithmus von Iliopoulos
3	$mgcd_{opt}$	Optimierter, linear, iterativer Algorithmus
4	$mgcd_{Blankin}$	Algorithmus von Blankinship
5	$mgcd_{BestRemainder}$	$mgcd_{Blankin}$ + best–remainder Strategie
6	$mgcd^{Pivot_1}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_1$
7	$mgcd^{Pivot_2}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_2$
8	$mgcd^{Pivot_3}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_3$
9	$mgcd^{Pivot_4}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_4$
10	$mgcd^{Pivot_{1+2}}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_{1+2}$
11	$mgcd^{Pivot_{1+4}}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_{1+4}$
12	$mgcd^{Pivot_{2+1}}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_{2+1}$
13	$mgcd^{Pivot_{2+3}}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_{2+3}$
14	$mgcd^{Pivot_{2+4}}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_{2+4}$
15	$mgcd^{Pivot_{3+2}}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_{3+2}$
16	$mgcd^{Pivot_{3+4}}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_{3+4}$
17	$mgcd^{Pivot_{4+1}}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_{4+1}$
18	$mgcd^{Pivot_{4+2}}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_{4+2}$
19	$mgcd^{Pivot_{4+3}}_{BestRemainder}$	$mgcd_{BestRemainder} + Pivot_{4+3}$
20	$mgcd_{Storjohann}$	Algorithmus von Storjohann
21	$mgcd_{Heuristik}$	Algorithmus mit gcd–Heuristik

Tabelle 10.60: Zusammenstellung mgcd–Algorithmen

Somit ergeben sich für den Kernalgorithmus HNF_{Z1} 22 und für den Kernalgorithmus HNF_{Z2} 132 verschiedene Algorithmen zur Berechnung der HNF.

Zur Verbesserung der Anschaulichkeit unterteilen wir die Gesamtheit der Algorithmen gemäß des verwendeten *mgcd*–Modulalgorithmus in 4 Algorithmengruppen, wie in Tabelle 10.61 dargestellt. In der aktuellen Implementierung äußert sich diese Gruppenbildung in der Namensgebung der verwendeten Interfacefunktionen.

Algorithmengruppe	mgcd–Kennzahlen
Hermite	0,1,2,3
Havas	4-19
Storjohann	20
gcd–Heuristik	21

Tabelle 10.61: Algorithmengruppen

10.1.4 Laufzeitenverhalten der HNF–Algorithmen

Das Laufzeitverhalten der nichtmodularen HNF–Algorithmen, die sich der zeilenweisen Elimination bedienen, wird durch das Laufzeitverhalten der verwendeten Modulalgorithmen *mgcd* und *normalize_matrix* dominiert. Die für diese Modulalgorithmen ermittelten Kenngrößen liefern uns eine hinreichend detaillierte Datenbasis, um Rückschlüsse auf das Laufzeitverhalten der jeweiligen HNF–Algorithmen ziehen zu können.

10.2 Spaltenweise Berechnung der HNF

Im speziellen Umfeld der Klassengruppenberechnung, auf das wir zu einem späteren Zeitpunkt noch eingehen werden, interessieren wir uns erst in zweiter Linie für die vollständige HNF einer dünnbesetzten, ganzzahligen Matrix. In erster Linie sind wir lediglich an den Spalten der HNF interessiert, die einen Diagonaleintrag größer als 1 besitzen. Aufgrund von empirischen Beobachtungen und den Resultaten aus [Dü91], [Jr.99] und [Nei01] wissen wir, daß diese Spalten mit hoher Wahrscheinlichkeit einigen wenigen der ersten Nicht–Nullspalten der HNF der Matrix entsprechen. Anders gesagt, die Diagonaleinträge, die etwas zur Gitterdeterminante des durch die Matrix erzeugten Gitters beitragen, konzentrieren sich auf den vorderen Teil der HNF. Diesen Teil der HNF bezeichnen wir im weiteren Verlauf dieses Abschnitts als den **Essentiellen Teil der HNF**.

Vor diesem Hintergrund haben wir uns mit der Frage beschäftigt, ob es möglich ist, für eine große, dünnbesetzte, ganzzahlige Matrix die HNF spaltenweise von links nach rechts zu berechnen.

Auf der Grundlage der Idee von Johannes Buchmann, das Problem, eine Spalte der HNF zu berechnen, auf das Problem, eine Lösung eines linearen Gleichungssystems über $\mathbb{Z}$ zu finden, zu reduzieren, ist es uns gelungen, Algorithmen zur Berechnung der HNF und insbesondere zur Berechnung des essentiellen Teils der HNF zu entwerfen, die das gewünschte leisten.

Diese Algorithmen bilden eine eigene Algorithmenklasse und sind Gegenstand dieses Abschnitts.

Vorgehensweise:
Die Algorithmen arbeiten in drei Phasen:

> **Phase 1:** Der Algorithmus startet mit der Matrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$.Der Einfachheit halber nehmen wir an, daß A vollen Zeilenrang besitzt. In der ersten Phase berechnen wir zunächst eine reguläre, quadratische Teilmatrix E und bringen diese mittels geeigneter Spaltentauschoperationen ans Ende der Matrix, so daß $A = (C|E)$ mit $\det_{\mathbb{Z}}(E) \neq 0$ gilt.
>
> **Phase 2:** In der zweiten Phase eliminieren wir schrittweise die Spalten der Teilmatrix C. Um die erste Spalte der Matrix C zu eliminieren, berechnen wir zuerst eine Lösung (x, k) mit $x \in \mathbb{Z}^m, k \in \mathbb{Z}$ und $\gcd(x, k) = 1$ des linearen Gleichungssystems
>
> $$E \cdot x = k \cdot c_{*,0}.$$

Anschließend betten wir die gefundene Lösung geeignet in einen Vektor $y = (-k, 0, \ldots, 0, x) \in \mathbb{Z}^n$ ein, ergänzen ihn zu einer unimodularen Matrix T mit $t_{*,0} = y$ und aktualisieren die Matrix $A = A \cdot T$. Dann gilt: $A = (0|C'|E')$ mit $E' \in \mathrm{GL}_m(\mathbb{Z})$ und $C' \in \mathbf{Mat}_{m\times(n-m-1)}(\mathbb{Z})$. Nun gehen wir rekursiv vor und eliminieren die erste Spalte der Matrix C'.

Phase 3: Nach der zweiten Phase sind wir in der Situation, daß wir alle Spalten der Matrix C eliminiert haben und lediglich eine Matrix $\hat{E} \in \mathrm{GL}_m(\mathbb{Z})$ als Träger der Gesamtinformation zurückbleibt. In der dritten Phase transformieren wir diese in HNF, indem wir eine Teilmatrix $\tilde{F}$ von $\hat{E}$ konstruieren mit

$$\tilde{F} \;=\; \begin{pmatrix} \hat{e}_{1,*} \\ \vdots \\ \hat{e}_{m-1,*} \end{pmatrix}$$

und anschließend mittels des in Phase 2 benutzten Verfahrens eine Matrix $T \in \mathrm{GL}_m(\mathbb{Z})$ berechnen, so daß gilt $\tilde{F} \cdot T = (0|\tilde{F}')$ mit $\tilde{F}' \in \mathrm{GL}_{m-1}(\mathbb{Z})$. Die erste Spalte der HNF der anfänglichen Matrix A errechnet sich nun durch

$$E \cdot T \;=\; \begin{pmatrix} h & * & \ldots & * \\ 0 & * & \ldots & * \\ \vdots & \vdots & \ddots & \vdots \\ 0 & * & \ldots & * \end{pmatrix}$$

Ist der Eintrag h negativ, negieren wir die Einträge der gesamten Spalte. Rekursiv können wir nun weitere Spalten der HNF berechnen.

Zur Veranschaulichung haben wir die verwendete Vorgehensweise nochmals in den folgenden Abbildungen dargestellt.

Schema:

- **HNF–Berechnung: Phase 1 + 2**

$$\begin{pmatrix} * & \ldots & * & * \\ \vdots & \ddots & \vdots & \vdots \\ * & \ldots & * & * \end{pmatrix} \rightarrow \begin{pmatrix} 0 & * & \ldots & * \\ \vdots & \vdots & \ddots & \vdots \\ 0 & * & \ldots & * \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 0 & * & \ldots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & * & \ldots & * \end{pmatrix}$$

- **HNF–Berechnung: Phase 3**

$$\begin{pmatrix} * & \ldots & * & * \\ \vdots & \ddots & \vdots & \vdots \\ * & \ldots & * & * \\ * & \ldots & * & * \end{pmatrix}$$

$$\downarrow$$
$$\begin{pmatrix} * & * & \dots & * \\ 0 & * & \dots & * \\ \vdots & \vdots & \ddots & \vdots \\ 0 & * & \dots & * \end{pmatrix}$$
$$\downarrow$$
$$\begin{pmatrix} * & * & * & \dots & * \\ 0 & * & * & \dots & * \\ 0 & 0 & * & \dots & * \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & * & \dots & * \end{pmatrix}$$
$$\vdots$$
$$\begin{pmatrix} 0 & * & \dots & * & * \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & * & * \\ 0 & \dots & 0 & 0 & * \end{pmatrix}$$

Grundalgorithmus:

Gemäß der obigen Beschreibung bildet somit der folgende Kernalgorithmus die Basis der in diesem Abschnitt dargestellten Algorithmenklasse.

10.44. Algorithmus

Kernalgorithmus zur spaltenweisen HNF–Berechnung: HNF_{GLS}

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbf{Z})$ mit $\mathrm{rank}_{\mathbf{Z}}(A) = m$.
AUSGABE: $\mathrm{HNF}(A)$.

[Bestimmung einer regulären Teilmatrix]
(1) $v = \mathbf{lininc}_{\mathbf{Z}}(\mathbf{A})$;
(2) $T = regular\_sort(v)$;
(3) $A = A \cdot T$; // $A \cdot T = (C|E)$
(4) $A.divide\_h(C, E)$; // $det_{\mathbf{Z}}(E) \neq 0$
(5) $A = \mathbf{update\_relations}(\mathbf{A}, \mathbf{T})$;
[HNF–Berechnung Phase 1]
(6) **for** $(i = 0; i < n - m; i++)$ **do**
(7) $b = a_{*,i}$;
(8) $(x, k) = \mathbf{solve}_{\mathbf{Z}}(\mathbf{E}, \mathbf{b})$; // $E \cdot x = k \cdot b$
(9) $y = (0, \dots, 0, -k, 0, \dots, 0, x)$;
(10) $y = \frac{y}{\gcd(c)}$;
(11) $A = \mathbf{update\_relations}(\mathbf{A}, \mathbf{basis\_completion}(\mathbf{y}))$;
(12) **od**

[HNF–Berechnung Phase 2]

(13) **for** $(i = 0; i < n; i++)$ **do**

(14) $b = \begin{pmatrix} e_{i+1,i} \\ \vdots \\ e_{n-1,i} \end{pmatrix}$;

(15) $E' = \begin{pmatrix} e_{i+1,i+1} & \cdots & e_{i+1,n-1} \\ \vdots & \ddots & \vdots \\ e_{n-1,i+1}) & \cdots & e_{n-1,n-1} \end{pmatrix}$;

(16) $(x, k) = \mathbf{solve}_{\mathbb{Z}}(\mathbf{E'}, \mathbf{b})$; // $E' \cdot x = k \cdot b$

(17) $y = (0, \ldots, 0, -k, 0, \ldots, 0, x)$;

(18) $y = \frac{y}{\gcd(x)}$;

(19) $A = \mathbf{update\_relations}(\mathbf{A}, \mathbf{basis\_completion}(\mathbf{y}))$;

(20) **od**

(21) **return** (A);

10.45. Bemerkung
In dem obigen Kernalgorithmus fällt auf, daß wir abweichend von der bisher gewählten Notation die Aufrufe der Modulalgorithmen *update_realtions* und *basis_completion* ineinander verschachtelt haben. Dies deutet darauf hin, daß der Modulalgorithmus *basis_completion* in verschiedenen Ausprägungen verschiedene Rückgabewerte besitzt und dementsprechend verschiedene Implementierungen der Funktion *update_relations* existieren.

Desweiteren wollen wir an dieser Stelle anmerken, daß die Matrix A in obigem Kernalgorithmus zum einem als einfache Matrix anzusehen ist, und zum anderen als eine Matrix in Potenzproduktdarstellung aufgefaßt werden kann. Somit haben wir es nur vordergründig mit *einem* Kernalgorithmus zu tun. In LiDIA wurden beide Varianten implementiert.

Als Modulalgorithmen identifizieren wir in diesem Kernalgorithmus

- $lininc_{\mathbb{Z}}(A)$ zur Berechnung der Indizes der linear unabhängigen Spalten der Matrix A,
- $solve_{\mathbb{Z}}(A, b)$ zur Berechnung einer Lösung über $\mathbb{Z}$ des linearen Gleichungssystems $A \cdot x = b$,
- $basis\_completion(x)$ zur Ergänzung eines Vektors $x \in \mathbb{Z}^n$ mit $\gcd(x) = 1$ zu einer Basis von $\mathbb{Z}^n$ und
- $update\_relations(A, T)$ um die Matrix A mittels der unimodularen Transformationsmatrix T zu aktualisieren.

Die Funktion *regular_sort* erzeugt aus einem Indexvektor $v \in \mathbb{Z}^n$ eine unimodulare Transformationsmatrix T, so daß gilt: $A \cdot T = (C|E)$ mit $E \in \mathbf{Mat}_{m \times m}(\mathbb{Z})$ und $\det_{\mathbb{Z}}(E) \neq 0$.

In den folgenden Abschnitten gehen im Detail auf verschiedene Möglichkeiten ein, diese Modulalgorithmen zu realisieren.

10.2.1 Berechnung der linear unabhängigen Spalten

Die Berechnung der linear unabhängigen Spalten einer ganzzahligen Matrix A gehört zu den Standardproblemen der Linearen Algebra über $\mathbb{Z}$. Die einfachste Art und Weise, diese Informationen zu berechnen, beruht darauf, die Eingabematrix A durch unimodulare Zeilenoperationen in eine obere Dreiecksmatrix zu transformieren und anhand der entstandenen Spaltenstufenform die Indizes der linear unabhängigen Spalten abzulesen. Wie bereits zu Beginn dieser Arbeit ausgeführt, erhöht sich bei dieser Vorgehensweise zunächst die Eintragsdichte und führt schließlich zu einer Explosion der Zwischenergebnisse.

Um effizient die linear unabhängigen Spalten berechnen zu können, müssen wir also auch hier Methoden und Techniken anwenden, die erstens die Dünnbesetztheit der Eingabematrizen weitestgehend erhalten und zweitens die *entry explosion* verhindern bzw. mindern.

Gemäß den Betrachtungen in den Arbeiten [Mül94] und [The95] und unseren Ausführungen im Abschnitt 2.4 können wir das Problem, die über $\mathbb{Z}$ linear unabhängigen Zeilen bzw. Spalten einer ganzzahligen Matrix A zu berechnen, auf äquivalente Probleme in endlichen Primkörpern übertragen. Dabei arbeiten wir gemäß der folgenden Vorgehensweise:

1. Zunächst berechnen wir die Hadamard–Schranke $H = \text{Hadamard}(A)$ der Matrix A.
2. Anschließend bestimmen wir eine Liste von Primzahlen, deren Produkt größer als das Doppelte der Hadamard–Schranke der Matrix A ist.
3. Nun berechnen wir für jeden zu einer Primzahl zugehörigen Primkörper die über diesem Primkörper linear unabhängigen Zeilen bzw. Spalten der Matrix A.
4. Abschließend vergleichen wir die erhaltenen Lösungen und geben die Lösung eines Primkörpers als Lösung über $\mathbb{Z}$ zurück, die maximal viele linear unabhängige Zeilen bzw. Spalten besitzt.

Wir stehen somit vor der Aufgabe, die linear unabhängigen Spalten einer Matrix A in einem endlichen Primkörper $\mathbb{F}_p$ zu berechnen. Folgende Methoden sind bekannt und wurden entsprechend implementiert:

1. **Gaußelimination**
   Die einfachste Methode, die über $\mathbb{F}_p$, $p \in \mathbb{P}$ linear unabhängigen Zeilen und Spalten einer Matrix A zu berechnen, besteht darin, die Matrix A in eine obere bzw. untere Dreiecksmatrix mittels unimodularer Zeilen- bzw. Spaltenoperationen zu transformieren und anschließend aus der entstandenen Spaltenstufenform die Indizes der gesuchten Zeilen bzw. Spalten abzulesen [Mül94, The95].
2. **Gaußelimination mit Pivotierung**
   Die oben vorgestellten Algorithmen arbeiten effizient, nutzen aber nicht die Dünnbesetztheit der Eingabematrix aus. Es liegt nun die Idee nahe, analog zu den Überlegungen zur zeilenweisen HNF–Berechnung Pivottechniken zu verwenden, um Varianten der Gaußelimination zu entwickeln, die besser die geringe Dichte der Eingabematrix zur Effizienzsteigerung ausnutzen. Eine detaillierte Untersuchung ist nicht Gegenstand dieser Arbeit.

10.2.2 Gleichungssystemlöser

In diesem Abschnitt gehen wir auf Algorithmen zur Berechnung einer Lösung bestehend aus einem Vektor x und einem Koeffizienten k des linearen Gleichungssystems

$$A \cdot x = k \cdot b$$

mit $A \in \mathrm{GL}_n(\mathbb{Z})$ und $b \in \mathbb{Z}^n$ ein.

Aufgrund der besonderen Eigenschaften des linearen Gleichungssystems wissen wir, daß mit $k = det_{\mathbb{Z}}(A)$ eine eindeutig bestimmte Lösung $x = adj_{\mathbb{Z}}(A) \cdot b$ existiert. Wollen wir uns diese Eigenschaft zunutze machen, so stehen wir auf der einen Seite vor dem Problem, die Determinante der Matrix A zu berechnen, und auf der anderen Seite interessieren wir uns für Algorithmen, die die adjungierte Matrix bestimmen oder das entstandene Gleichungssystem

$$A \cdot x \quad = \quad \det_{\mathbb{Z}}(A) \cdot b$$

lösen.

10.2.2.1 Determinantenberechnung

Der naive Algorithmus, die Determinante einer ganzzahligen Matrix zu berechnen, transformiert die Eingabematrix A in eine obere Dreiecksmatrix und berechnet anschließend das Produkt der Diagonalelemente. Dieser Algorithmus trägt weder der dünnbesetzten Struktur der Eingabematrix Rechnung noch bekämpft er effizient die auftretende *entry explosion*. Damit ist der zur Durchführung benötigte Speicherplatzbedarf hoch und der Algorithmus in Bezug auf die Laufzeit ineffizient.

Eine Möglichkeit die *entry explosion* zu bekämpfen liegt wiederum, wie bereits in Abschnitt 2.4 erwähnt, in der Anwendung des Mechanismus der modularen Arithmetik. Dieser Mechanismus ist auf die Problemstellung, die Determinante einer ganzzahligen Matrix zu berechnen, anwendbar, weil folgendes gilt:

1. **Schranke**
   Die Hadamard–Schranke der Eingabematrix A ist eine obere Schranke für die Determinante, d.h.:

   $$\mathrm{Hadamard}(A) \geq |\det_{\mathbb{Z}}(A)|$$

2. **Übertragbarkeit**
   Für alle Primzahlen $p \in \mathbb{P}$ gilt:

   $$\det_{\mathbb{Z}}(A) \equiv \det_{\mathbb{F}_p}(A) \bmod p$$

Somit sind wir in der Lage mittels des Chinesischen Restsatzes die Problemstellung, die Determinante über $\mathbb{Z}$ zu berechnen, auf adäquate Probleme über endlichen Primkörpern zu verlagern und somit *entry explosion* von vorne herein auszuschließen. Die Vorgehensweise entspricht der in Abschnitt 2.4 dargestellten.

10.46. Bemerkung
Für komplexitätstheoretische Betrachtungen kann man annehmen, daß man die Berechnung in einem endlichen Körper hinreichend großer Charakteristik durchführt und sich somit die Anwendung des Chinesischen Restsatzes spart. Für die Praxis ist es im Allgemeinen günstiger, die Berechnung auf viele endliche Körper zu verteilen, die wenn möglich mit einfacher Präzision realisiert sind, und das gewünschte Ergebnis mittels des Chinesischen Restsatzes zu ermitteln (vgl. [The95]).

Wir stehen nun vor der Aufgabe, die Determinante einer Matrix über einem endlichen Primkörper $\mathbb{F}_p$ zu berechnen. Es bieten sich dazu die folgenden Methoden an:

1. **Gauß–Bareiss**
   Die einfachste Methode, die Determinante einer Matrix über einem endlichen Körper zu berechnen, besteht darin, die Matrix mittels geeigneter unimodularer Operationen in eine obere Dreiecksmatrix zu transformieren und anschließend das Produkt der Diagonalelemente zu bilden. Der Algorithmus wurde von Bareiss dahingehend erweitert, daß nur diejenigen Elemente einer oberen Dreiecksmatrix berechnet werden, die einen Beitrag zur Determinante leisten [Coh93].

   Der Nachteil dieser Methode liegt wiederum darin, daß dieser Algorithmus die Dünnbesetztheit der Eingabematrix nicht explizit zur Effizienzsteigerung ausnutzt. Wiederum kann eine geeignete Pivotstrategie dazu dienen, die Dünnbesetztheit während der Berechnung weitgehend zu erhalten und somit die Ausführungseffizienz zu erhöhen.

   Auf eine detaillierte Algorithmusbeschreibung verzichten wir an dieser Stelle und verweisen auf [Coh93].

2. **Wiedemann–Algorithmus**
   Douglas H. Wiedemann hat in seiner Arbeit [Wie86] einen probabilistischen Algorithmus vom Typ Las Vegas vorgestellt, der explizit die Dünnbesetztheit einen $(m \times m)$–Eingabematrix A zur Berechnung der Determinanten $\det_{\mathbb{F}_p}(A)$ über einem endlichen Primkörper mit einer Charakteristik $p > 50 \cdot m^2 \log_2(m)$ ausnutzt.

   Wiederum verzichten wir an dieser Stelle auf eine detaillierte Beschreibung des Algorithmus und verweisen auf die Originalarbeit [Wie86].

Gemäß der beiden Möglichkeiten, die Determinante in einem endlichen Primkörper zu berechnen, stehen uns somit auch zwei Möglichkeiten zur Berechnung der Determinante über $\mathbb{Z}$ zur Verfügung.

Es ist leicht einzusehen, daß die Laufzeit jeweils stark von der Anzahl und Charakteristik der benötigten Primkörper abhängt. Aufgrund dieser Beobachtung liegt die folgende heuristische Erweiterung auf der Hand.

CRT–Heuristik
Es ist bekannt, daß die Hadamard–Schranke im allgemeinen eine schlechte Approximation an die reale Determinante einer regulären, quadratischen, ganzzahligen Matrix A darstellt. Dies führt zu der Feststellung, daß viele Determinantenberechnungen in endlichen Körpern keine neue Information zur Berechnung der Determinante über $\mathbb{Z}$ beitragen, weil

die reale Determinante deutlich kleiner als die Hadamard–Schranke der Matrix ist und somit bereits berechnet wurde. Auf dieser Beobachtung basiert der folgende heuristische Algorithmus. Mittels eines zusätzlichen Eingabeparameters $b \in \mathbb{N}$ entscheidet der Algorithmus, ob ein weiterer endlicher Körper zur Berechnung der Determinanten herangezogen wird oder nicht. Dazu protokolliert er mittels eines Zählers, wie oft aufeinanderfolgende Durchführungen des Chinesichen Restsatzes (Chinese Remainder Theorem – CRT) bereits das gleiche Ergebnis geliefert haben, d.h. wie oft bereits Determinantenberechnungen in endlichen Körpern keine weitere Information zur Determinanten über $\mathbb{Z}$ beigetragen haben. Ist der aktuelle Zählerstand größer als der Parameter b, wird die Berechnung beendet. Im anderen Fall wird die Berechnung fortgesetzt. Im schlimmsten Fall müssen alle endlichen Körper, die durch die Primzahlliste bzw. durch die Hadamard–Schranke vorgegeben sind, zur Determinantenberechnung herangezogen werden.

In diesem Algorithmus können sowohl der Algorithmus von Gauß–Bareiss als auch der Algorithmus von Wiedemann als Modulalgorithmus zur Berechnung der Determinante in einem endlichen Primkörper benutzt werden.

10.47. Algorithmus

Heuristische Variation zur Determinantenberechnung: $\det_{CRT}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ mit $\text{rank}_{\mathbb{Z}}(A) \neq m$ und $b \in \mathbb{N}$.
AUSGABE: HNF(A).

```
     [Initialisierung]
(1)    H = Hadamard(A);
(2)    (l, p) = get_primes(2 · H);          // Primzahlliste p der Länge l
(3)    z = 1; m = 1;
     [Determinantenberechnung]
(4)    for (i = 0; i < l; i++) do
(5)      d' = det_{F_{p_i}}(A);
(6)      if (i > 0) then
(7)        d' = chinrest(d, m, d', p_i);
(8)        m = m · p_i;
(9)        if (d' = d) then
(10)         z++;
(11)       else
(12)         d = d';
(13)         z = 1;
(14)       fi
     [Short-Cut]
(15)       if (z > b) then
(16)         return (d);
(17)       fi
(18)     else
(19)       d = d';
(20)     fi
```

```
(21)  od
(22)  return (d);
```

Die oben beschriebene Variation zur Determinantenberechnung kann bedingt durch die integrierte Heuristik auch falsche Ergebnisse liefern. Dies muß bei der Integration dieser Variation in andere Algorithmen und Abläufe beachtet werden.

10.2.2.2 Lösung des Linearen Gleichungssystems

Aufgrund der Voraussetzung, daß wir an der Hermite–Normalform großer, dünnbesetzter Matrizen interessiert sind, sind wir auf der Suche nach Gleichungssystemlösern über endlichen Primkörpern, die dieser speziellen Situation Rechnung tragen. Es bieten sich die folgenden Algorithmen an:

1. **Gaußelimination**
   Die Gauß–Elimination ist der bekannteste und verbreiteste Algorithmus, den man zur Berechnung einer Lösung eines linearen Gleichungssystems über Körpern heranziehen kann. Er besitzt den Nachteil, wie schon mehrfach erwähnt, daß er in der Originalversion nicht an dünnbesetzte Eingabematrizen angepaßt ist.

   Mittels geeigneter Pivotierungsstrategien kann man den Gaußalgorithmus an Eingabematrizen geringer Dichte anpassen.

2. **Wiedemann–Algorithmus**
   Der Algorithmus von Wiedemann berechnet, wie schon weiter oben erwähnt, eine Lösung eines linearen Gleichungssystems über einem endlichen Primkörper. Dieser Algorithmus ist im Detail in den Arbeiten [Wie86, BM91, LO91] beschrieben.

3. **Lanczos–Algorithmus**
   Ein weiterer Algorithmus zur Berechnung einer Lösung eines linearen Gleichungssystems über endlichen Primkörpern ist der Lanczos–Algorithmus. Es handelt sich dabei um einen sog. Krylo–Subspace–Algorithmus. Auf eine genaue Beschreibung dieses Algorithmus und seiner Eigenschaften verzichten wir an dieser Stelle und verweisen auf die Arbeiten [Den97, Lan52, Mon95].

4. **Konjugierte Gradienten–Methode**
   Die Konjugierte Gradienten–Methode ist der letzte Algorithmus, den wir in unsere Betrachtungen einbeziehen. Dieser Algorithmus stammt aus dem Bereich der Linearen Optimierung. Wie bereits bei den vorangegangenen Algorithmen, verzichten wir in dieser Arbeit auf eine genaue Beschreibung dieses Algorithmus und verweisen auf verschiedene Referenzarbeiten, die den Algorithmus im Detail vorstellen und die uns als Grundlage für die Implementierung gedient haben. Für die Konjugierte Gradienten–Methode sind dies die Arbeiten [BM91] und [LO91].

Uns stehen somit vier Algorithmen zur Berechung einer Lösung eines linearen Gleichungssystems in endlichen Primkörpern zur Verfügung.

10.2.3 Basisergänzung

Ausgehend von der Kenntnis eines Lösungsvektors $x \in \mathbb{Z}^n$ mit $\gcd(x) = 1$ betrachten wir in diesem Abschnitt Algorithmen, die diesen Vektor zu einer Basis des $\mathbb{Z}^n$ ergänzen, sprich eine Matrix $T \in \mathrm{GL}_n(\mathbb{Z})$ konstruieren mit $x = t_{*,0}$.

In den Arbeit [Nei94] wird der folgende Algorithmus vorgeschlagen, der das gewünschte leistet.

10.48. Algorithmus

Algorithmus zur Basisergänzung: $basis\_completion_{Normal}$

EINGABE: $v \in \mathbb{Z}^n$ mit $\gcd(v) = 1$.
AUSGABE: $T \in \mathrm{GL}_n(\mathbb{Z})$ mit $t_{*,0} = v$.

[Initialisierung]
(1) **for** (i = 0; i < n; i++) **do**
(2) $y_i = v_i$;
(3) **od**
(4) $y_n = -1$;
(5) **for** (i = 0; i < n; i++) **do**
(6) $b_{*,i} = e_i^n$;
(7) **od**
(8) $b_{*,n} = v$
[Basisergänzung]
(9) **for** $(i = 0; i < n; i++)$ **do**
(10) $g = xgcd(y_{i+1}, y_i, a, b)$;
(11) $(b_i, b_{i+1}) = (b_i, b_{i+1}) \begin{pmatrix} -\frac{y_i}{g} & a \\ \frac{y_{i+1}}{g} & b \end{pmatrix}$;
(12) $y_{i+1} = g$;
(13) $y_i = 0$;
(14) **if** $(y_{i+1} = 1)$ **then**
(15) $T = (v, b_1, \ldots, b_i, b_{i+2}, \ldots, b_n)$;
(16) **fi**
(17) **od**
(18) **return** (T);

Betrachtet man diesen Algorithmus genauer, so stellt man fest, daß die Dichte der berechneten unimodularen Matrix T direkt mit der Anzahl der Elemente des Lösungsvektors (beginnend beim ersten Element) korelliert, die als größten gemeinsamen Teiler ± 1 besitzen. Je weniger Elemente benötigt werden, um eine Zerlegung von ± 1 zu konstruieren, umso weniger Elemente besitzt die unimodulare Matrix.

Aus den Arbeiten von Storjohann [Sto96b, Sto96a] wissen wir, daß es uns mittels Konditionierung möglich ist, die Berechnung des größten gemeinsamen Teilers einer Menge von Zahlen auf die Berechnung des größten gemeinsamen Teilers zweier Zahlen zu reduzieren.

Kombiniert man die Idee der Konditionierung mit dem obigen Algorithmus, erhält man den folgenden Algorithmus zur Basisergänzung.

10.49. Algorithmus

Algorithmus zur Basisergänzung: $basis\_completion_{cond}$

EINGABE: $v \in \mathbb{Z}^n$ mit $\gcd(v) = 1$.
AUSGABE: $C^{-1}, T \in \mathrm{GL}_n(\mathbb{Z})$ mit $t'_{*,0} = v$ und $T' = C^{-1} \cdot T$.

(1) $C = cond\_matrix(v)$;
(2) $v = C \cdot v$;
(3) $g = xgcd(v_0, v_1, a, b)$;
(4) $T = I_n$;
(5) $t_{*,0} = v$;
(6) $t_{0,1} = -a$;
(7) $t_{1,1} = b$;
(8) **return** (C^{-1}, T);

Der obige Algorithmus nutzt somit die Eigenschaft $v = C^{-1} \cdot C \cdot v$ aus, indem zunächst $C \cdot v$ zu einer unimodularen Matrix ergänzt und anschließend die Konditionierungsmatrix C durch die Bildung der inversen Matrix C^{-1} eliminiert wird. Bemerkenswert ist, daß die Berechnung der inversen Matrix aufgrund der besonderen Struktur der Konditionierungmatrix

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & c_0 & c_1 & \dots & c_{n-3} \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & & 0 \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

einfach ist. Wie man leicht durch Multiplikation verifizieren kann, gilt:

$$C^{-1} = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & -c_0 & -c_1 & \dots & -c_{n-3} \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & & 0 \\ \vdots & \vdots & \vdots & & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

10.2.4 Aktualisierung der Matrix

Für die Aktualisierung einer dünnbesetzten Matrix mittels unimodularer Matrizen bieten sich die folgenden beiden Möglichkeiten an, die beide in LiDIA integriert sind und somit verwendet werden.

1. **Multiplikation**
   Die einfachste Methode, in einer Matrix $T \in \mathrm{GL}_n(\mathbb{Z})$ gespeicherte unimodulare Spaltenoperationen auf eine Matrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ anzuwenden, besteht darin, T von rechts mit A zu multiplizieren.

   Die Aktualisierungsfunktion hat dann das folgende Aussehen:

10.50. Algorithmus

Aktualisierung durch Multiplikation: $update\_relations_{Mult}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ und $T \in \mathrm{GL}_n(\mathbb{Z})$.
AUSGABE: $A \cdot T$

(1) **return** $(A \cdot T)$;

   Die Erweiterung für zwei Eingabematrizen ist naheliegend und ergibt den folgenden Algorithmus:

10.51. Algorithmus

Aktualisierung durch Multiplikation: $update\_relations_{Mult}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ und $T_1, T_2 \in \mathrm{GL}_n(\mathbb{Z})$.
AUSGABE: $A \cdot T_1 \cdot T_2$

(1) **return** $(A \cdot T_1 \cdot T_2)$;

   Der Vorteil dieser Aktualisierungsprozedur liegt in der Einfachheit der Funktion und der damit leichten Programmierbarkeit und Verständlichkeit. Ein entscheidender Nachteil liegt darin, daß die Multiplikation zweier dünnbesetzter Matrizen im allgemeinen eine dichtbesetzte Matrix ergibt.

2. **Potenzproduktdarstellung**
   Im Gegensatz zu den Algorithmen, die sich der zeilenweisen Elimination (siehe Abschnitt 10.1) bedienen, müssen die Algorithmen dieser Matrixklasse nicht explizit auf Elemente einer Zeile der Matrix zugreifen. Desweiteren benötigen einige der Algorithmen zur Berechnung einer Lösung eines linearen Gleichungssystems lediglich die Möglichkeit, einen Vektor von rechts mit der Matrix multiplizieren zu können.

Aus diesem Grund ist es sinnvoll, die Verwendung einer Potenzproduktdarstellung für Matrizen in Betracht zu ziehen. Dementsprechend ergeben sich die folgenden beiden Algorithmen zur Aktualisierung der Matrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ mittels einer unimodularen Matrix $T \in \mathrm{GL}_n(\mathbb{Z})$.

10.52. Algorithmus

Aktualisierung durch Multiplikation:
update_relations$_{PowerProdukt}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ in Potenzproduktdarstellung und
$T \in \mathrm{GL}_n(\mathbb{Z})$.
AUSGABE: A in Potenzproduktdarstellung mit
$A.compute() = A \cdot T$

```
(1)  A.append(T);                    // A = (A, T)
(2)  return (A);
```

Wiederum ist die Erweiterung auf zwei Eingabematrizen leicht und man erkennt, aus welchem Grund die Verschachtelung der Modulalgorithmen *update_relations* und *basis_completion* im Kernalgorithmus sinnvoll ist.

10.53. Algorithmus

Aktualisierung durch Multiplikation:
update_relations$_{PowerProdukt}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ in Potenzproduktdarstellung und
$T_1, T_2 \in \mathrm{GL}_n(\mathbb{Z})$.
AUSGABE: A in Potenzproduktdarstellung mit
$A.compute() = A \cdot T_1 \cdot T_2$

```
(1)  A.append(T_1);                  // A = (A, T_1)
(2)  A.append(T_2);                  // A = (A, T_2)
(3)  return (A);
```

10.54. Bemerkung [*LiDIA*]
In LiDIA wurde die Möglichkeit, Potenzprodukte zu verwalten und mit Potenzprodukten zu rechnen, in einer generischen Weise implementiert. Die Template–Klasse *power_product* bietet die Möglichkeit, aus einer Matrix die Matrix in Potenzproduktdarstellung zu erzeugen und umgekehrt aus einer Matrix in Potenzproduktdarstellung eine Matrix zu berechnen, die dem Produkt der in der Potenzproduktdarstellung enthaltenen Matrizen entspricht. Letzteres wird durch die Member–Funktion *compute* realisiert. Die Member–Funktion *append* fügt eine Matrix an die Potenzproduktdarstellung an.

10.2.5 Algorithmenübersicht

Die Algorithmenklasse der spaltenweisen Berechnung der HNF besteht aus einem Kernalgorithmus (auf der Basis von zwei Matrixdarstellungen) kombiniert mit 2 Modulalgorithmen zur Berechnung der linear unabhängigen Spalten (Tabelle 10.62), 4 Modulalgorithmen zur Berechnung der Determinante (Tabelle 10.63), 4 Modulalgorithmen zur Berechnung einer Lösung eines linearen Gleichungssystems (Tabelle 10.64) und 2 Modulalgorithmen zur Basisergänzung (Tabelle 10.65).

Kennzahl	**lininc–Algorithmus**	**Bezeichnung**
0	$lininc_{Gauß}$	Gaußelimination
1	$lininc_{Gauß+Pivot}$	Gaußelimination mit Pivotierung

Tabelle 10.62: Zusammenstellung *lininc*–Algorithmen

Kennzahl	**det–Algorithmus**	**Bezeichnung**
0	$\det_{Gauß-Bareiss}$	Grundalgorithmus + Gauß–Bareiss
1	$\det_{Wiedemann}$	Grundalgorithmus + Wiedemann
2	$\det_{CRT+Gauß-Bareiss}$	CRT–Grundalgorithmus + Gauß–Bareiss
3	$\det_{CRT+Wiedemann}$	CRT–Grundalgorithmus + Wiedemann

Tabelle 10.63: Zusammenstellung *det*–Algorithmen

Kennzahl	**solve–Algorithmus**	**Bezeichnung**
0	$solve_{Gauß}$	Gaußelimination
1	$solve_{Wiedemann}$	Wiedemann Algorithmus
2	$solve_{Lanczos}$	Lanczos Algorithmus
3	$solve_{KonjugierteGradienten}$	Konjugierte Gradienten Methode

Tabelle 10.64: Zusammenstellung *solve*–Algorithmen

Somit umfaßt diese Algorithmenklasse kombinatorisch gesehen $2 \cdot 4 \cdot 4 \cdot 2 = 64$ Algorithmen in einer Matrixdarstellung, also insgesamt 128 Algorithmen.

10.2.6 Laufzeitverhalten der HNF–Algorithmen

Das Laufzeitverhalten der Algorithmen der hier betrachteten Algorithmenklasse wird zum einen durch die verwendeten Modulalgorithmen impliziert und zum anderen durch die Größe der Einträge der in jeder Iteration entstehenden unimodularen Matrizen bestimmt. Bedenkt man, daß in jeder Iteration zwei unimodulare Matrizen berechnet werden, die in der folgenden Iteration in die Berechnung der Lösung des linearen Gleichungssystems eingehen, so wird klar, daß von Iteration zu Iteration die Größe der Einträge des bestimmten Lösungsvektors ansteigt, damit größere Einträge in den daraus resultierenden unimodularen Matrizen entstehen und damit eine (implizite) Explosion der Einträge stattfindet.

Kennzahl	Basisergänzung–Algorithmus	Bezeichnung
0	$basis\_completion_{Normal}$	Naiver Basisergänzung
1	$basis\_completion_{cond}$	Basisergänzung mittels Konditionierung

Tabelle 10.65: Zusammenstellung *basis_completion*–Algorithmen

Im Rahmen von verschiedenen Testreihen haben wir festgestellt, daß aufgrund des oben beschriebenen Zusammenhangs die in diesem Abschnitt betrachteten Algorithmen einen hohen Rechenzeitbedarf an den Tag legen, während sich der Speicherplatzbedarf aufgrund der Möglichkeit, dünnbesetzte Matrizen in Potenzproduktdarstellung zu speichern, in Grenzen hielt.

Die Algorithmen dieser Algorithmenklasse sind somit lediglich dann geeignet, wenn nur wenige der ersten Spalten der HNF einer Matrix benötigt werden. Da wir uns in dieser Arbeit für die Berechnung der gesamten HNF einer Matrix interessieren, spielen die Algorithmen dieser Matrixklassen für den Fortgang der weiteren Arbeit keine Rolle.

10.55. Bemerkung
Auf eine detaillierte Betrachtung der Laufzeiteigenschaften der einzelnen Modulalgorithmen haben wir in dieser Arbeit verzichtet, da dies den Umfang der Arbeit bei weitem übersteigen würde.

10.3 Hauptminorenweise Berechnung der HNF

In diesem Abschnitt beschreiben wir eine dritte Klasse von Algorithmen zur Berechnung der HNF, die auf einer Vorgehensweise beruht, die von Kannan und Bachem entwickelt und in ihrer Arbeit [KB79] dargestellt wurde. Die Idee ihrer Vorgehensweise liegt darin, iterativ reguläre Untermatrizen (Hauptminoren) der eigentlichen Matrix auf Hermite–Normalform zu bringen.

10.56. Bemerkung
Die Originalbeschreibung der Vorgehensweise basiert auf der Definition der HNF als untere, linke Dreiecksmatrix. Daher stammt die Bezeichnung dieser Klasse.

Vorgehensweise:
Die Algorithmen dieser Algorithmenklasse beginnen in der unteren rechten Ecke der Eingabematrix und transformiert iterativ, eine wachsende quadratische Teilmatrix in HNF.

Anhand der folgenden schematischen Darstellung ist leicht zu sehen, wie die Algorithmen dieser Klasse arbeiten.

Schema:

$$\begin{pmatrix} * & \dots & * & * \\ \vdots & \ddots & \vdots & \vdots \\ * & \dots & * & * \\ * & \dots & * & * \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} * & \dots & * & * & * \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ * & \dots & * & * & * \\ * & \dots & * & 0 & * \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} * & \dots & * & * & * & * \\ \vdots & \ddots & \vdots & \vdots & \vdots & \vdots \\ * & \dots & * & * & * & * \\ * & \dots & * & 0 & * & * \\ * & \dots & * & 0 & 0 & * \end{pmatrix}$$

$$\vdots$$

$$\begin{pmatrix} 0 & * & \dots & * & * \\ \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & * & * \\ 0 & \dots & 0 & 0 & * \end{pmatrix}$$

Kernalgorithmus:

Die Algorithmen diese Algorithmenklasse basieren somit auf dem folgenden Kernalgorithmus.

10.57. Algorithmus

Kernalgorithmus nach Kannan und Bachem:
$\mathrm{HNF}_{KannanBachem}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$
AUSGABE: $\mathrm{HNF}(A)$

[Initialisierung]
(1) $T = I_n$
(2) $l = 2$;
[HNF–Berechnung]
(3) **while** $(l < n)$ **do**
(4) **for** $(j = 1; j < l; j++)$ **do**
(5) $p = n - j$;
[Handling für reguläre Teilmatrix]
(6) **if** $(a_{p,p} = 0)$ **then**

(7) **for** (i = p; i $\geq 0 \wedge a_{p,i} = 0$; i– –) **do**
(8) **od**
(9) $A.swap\_columns(i, p)$;
(10) **fi**
[Elimination]
(11) $g = xgcd(p, q, a_{p,n-l}, a_{p,p})$;
(12) $U = \begin{pmatrix} -\frac{a_{p,p}}{g} & p \\ \frac{a_{p,n-l}}{g} & q \end{pmatrix}$;
(13) $(a_{*,n-l}, a_{*,p}) = (a_{*,n-l}, a_{*,p}) \cdot U$;
(14) **od**
[Normalisierung]
(15) **A.normalize_matrix(p, p)**;
(16) **od**
(17) **return** (A;);

Wie man leicht aus dem obigen Algorithmus entnehmen kann, wird in jeder Iteration eine Normalisierung der aktuell betrachteten Teilmatrix durchgeführt. Gemäß Abschnitt 10.1.2 kennen wir verschiedene Modulalgorithmen, die dies leisten.

10.58. Bemerkung
Der im Originalpapier vorgestellte Algorithmus behandelt quadratische, nichtsinguläre, ganzzahlige Matrizen. Jedoch zeigt sich bei einer genaueren Betrachtung, daß eine offensichtliche Modifikation den Anwendungsbereich dieses Algorithmus auf Matrizen mit vollem Zeilenrang erweitert.

Der Vollständigkeit halber geben wir noch das folgende Lemma an.

10.59. Lemma
Ist eine quadratische Matrix A nichtsingulär, dann können die Spalten der Matrix A so permutiert werden, daß alle Hauptminoren der resultierenden Matrix nicht singulär sind.

Beweis: Siehe [KB79]. ■

10.3.1 Algorithmenübersicht

Diese Algorithmenklasse beinhaltet insgesamt lediglich 4 Algorithmen. Der oben angegebene Kernalgorithmus wird mit den in Tabelle 10.66 angegebenen Modulalgorithmen zur Normalisierung kombiniert.

Die anderen Normalisierungmodulalgorithmen sind nicht anwendbar, weil sie auf der Kenntnis der Gitterdeterminanten basieren, die uns erst am Ende der Berechnung zur Verfügung steht.

Kennzahl	Normalisierung	Bezeichnung
0	$Normalize_{Std}$	Standardnormalisierung
1	$Normalize_{ChouCollins}$	Normalisierung nach Chou–Collins
2	$NormalizeHybrid_{Std}$	Hybrid–Standardnormalisierung
3	$NormalizeHybrid_{ChouCollins}$	Hybrid–Normalisierung nach Chou–Collins

Tabelle 10.66: Zusammenstellung *normalize*–Algorithmen (Hauptminorenweise HNF–Berechnung)

10.3.2 Laufzeitverhalten der HNF–Algorithmen

Das Laufzeitverhalten der Algorithmen dieser Algorithmenklasse wird im Gegensatz zu den beiden vorangehenden Algorithmenklassen nicht durch das Laufzeitverhalten der verwendeten Modulalgorithmen dominiert. Die unterschiedlichen Normalisierungsroutinen wirken sich lediglich minimal auf die Laufzeit der Algorithmen aus. Im Detail wurden die zu dieser Algorithmenklasse gehörenden Algorithmen bereits in der Arbeit [Wag97] untersucht. In zusätzlichen Testreihen haben wir das Verhalten für dünnbesetzte Matrizen getestet.

Zusammenfassend kann man feststellen, daß die in diesem Abschnitt vorgestellten Algorithmen gut für dichtbesetzte, kleinere Matrizen geeignet sind. Für diese Matrizen übertreffen sie teilweise die erzielen Meßergebnisse für Algorithmen, die sich der zeilenweisen Eliminationsstrategie bedienen. Je geringer die Eintragsdichte der Startmatrix ist, umso mehr fallen die Laufzeit der Algorithmen dieser Algorithmenklasse hinter die der Algorithmen, die sich der zeilenweisen Eliminationsstrategie bedienen, zurück.

Kapitel 11

Modularen Algorithmen zur Berechnung der HNF

Die Grundidee aller Algorithmen, die wir in diesem Kapitel beschreiben, ist die Beobachtung, daß sich das von den Spalten einer Matrix $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ mit $\text{rank}(A) = m$ erzeugte Gitter $L(A)$ nicht ändert, wenn man $\alpha \cdot I_n$ mit A „konkateniert“. Hierbei bezeichnen wir mit α ein positives, ganzzahliges Vielfaches der Gitterdeterminanten von $L(A)$, d.h. des Volumens des von den Spalten der Matrix A aufgespannten Parallelotops. Es gilt also:

$$L(A) = L(A|\alpha \cdot I_n)$$

Aus dieser Eigenschaft resultiert die folgende Beobachtung, die zur Begrenzung der Größe der Zwischenresultate bei der Berechnung der HNF einer ganzzahligen Matrix A ausgenutzt werden kann:

$$\text{HNF}(A) = \text{HNF}(A|\alpha \cdot I_n) = \text{HNF}(A \bmod \alpha|\alpha \cdot I_n)$$

Hierbei steht α wiederum für ein ganzzahliges Vielfaches der Gitterdeterminanten von $L(A)$.

Wir können somit die Normalform einer Matrix A berechnen, indem wir die Normalform in $\mathbb{Z}/\alpha\mathbb{Z}$ berechnen und anschließend den implizit begangenen Fehlern korrigieren, indem wir die Diagonalmatrix $\alpha \cdot I_n$ zum Beispiel durch Gauß–Elimination entfernen [Mül94, Dom89, Dü91].

11.1. Bemerkung
Die Forderung $\text{rank}(A) = m$ ist ein essentiellen Bestandteil der Algorithmen dieser Algorithmenklasse. Eine Verallgemeinerung ist nur über geeignete Einbettung in ein “erweitertes Gitter” möglich.

Die oben beschriebene Vorgehensweise ging aus Untersuchungen zur Berechnung von Matrix–Normalformen über $\mathbb{Z}$ hervor und basiert auf der Arbeit von Domich [Dom89]. Der Vorteil dieser Vorgehensweise ist, daß die Größe der Zwischenresultate durch das Vielfache der Gitterdeterminanten α beschränkt ist. Demzufolge bedeutet dies, daß die Laufzeit der

Algorithmen, die sich dieser Vorgehensweise bedienen, neben der Anzahl der Elemente direkt und vorrangig von der Größe des Vielfachen der Gitterdeterminanten beeinflußt wird.

Um die Ausführungseffizienz der Implementierung zu verbessern, müssen wir unser Bestreben darauf ausrichten, ein möglichst kleines Vielfaches der Gitterdeterminanten zu finden.

Berechnung eines Vielfachen der Gitterdeterminanten

Ein erster Ansatz zur Berechnung eines Vielfachen der Gitterdeterminanten eines Gitters $L(A)$ besteht darin, eine maximale, reguläre Untermatrix von A zu suchen, deren Rang dem Rang von A entspricht, und von dieser Matrix die Determinante zu berechnen.

Die praktischen Resultate dieser Methode sind jedoch nicht überzeugend, d.h. die ermittelten Vielfachen sind weit von der Gitterdeterminanten des von den Spalten der Matrix A erzeugten Gitters $L(A)$ entfernt.

Deshalb wird dieser Ansatz wie folgt zu einer zweiten Methode erweitert. Anstatt nur eine maximale reguläre Untermatrix zu suchen, suchen wir mehrere und berechnen jeweils die Determinante. Als Vielfaches der Gitterdeterminanten wählen wir dann den größten gemeinsamen Teiler der berechneten Determinanten [HS79, The95].

Der Vorteil dieser Variante ist, daß die so erhaltenen Vielfachen wesentlich kleiner als die vorherigen sind. Sie hat aber den Nachteil, daß jede Berechnung einer regulären Untermatrix und der jeweiligen Determinanten relativ teuer im Verhältnis zur Gesamtberechnung ist. Deshalb stellen sich uns nun folgende Fragen:

1. Wieviele maximale reguläre Untermatrizen garantieren ein (annähernd) optimales Ergebnis?
2. Wieviel Zeit darf man in die Berechnung des Vielfachen der Gitterdeterminanten investieren, damit entsprechende Zeitvorteile in der folgenden Berechnung wirksam werden?

Die erste Fragestellung führt auf direktem Wege zu dem MINIMUM–GCD–SET–Problem, welches, wie schon des öfteren erwähnt, NP–vollständig ist [MH94b, MH94a].

Somit streben wir danach, gute Heuristiken zu finden, die eine für unsere Belange gute Lösung liefern.

Mit dieser Zielsetzung haben wir verschiedene Versuche mit einer unterschiedlichen Anzahl an zu wählenden Untermatrizen durchgeführt, die nach unterschiedlichen Konstruktionsschemata berechnet wurden. Dabei haben wir folgendes Ergebnis erhalten:

- **Methode 1:** Für nicht quadratische, zufällig erzeugte Matrizen mit Einträgen betragsmäßig kleiner als $2^{31}-1$ ist der „Trade of“ zwischen der Anzahl der Untermatrizen und der Größe des Vielfachen der Gitterdeterminanten optimal, wenn zunächst

zwei reguläre Untermatrizen, die möglichst wenige Einträge gemeinsam haben, gesucht werden, dann die Determinanten dieser erhaltenen Teilmatrizen berechnet werden und zuletzt der gcd über den Determinanten gebildet wird.

- **Methode 2:** Für quadratische Matrizen ist die Wahl nur einer regulären Untermatrix mit maximalem Rang optimal, weil in vielen Fällen die Determinante der quadratischen Matrix der Gitterdeterminanten entspricht, d.h. daß die quadratische Matrix regulär ist.

Aufgrund der Betrachtungen in den Abschnitten 10.2.1 und 10.2.2.1 sind uns bereits Algorithmen zur Berechnung des Rangs bzw. der linear unabhängigen Spalten einer beliebigen Matrix und der Determinanten einer quadratischen Matrix bekannt.

Somit stehen uns alle Informationen zur Verfügung, um die modularen Algorithmen zur Berechnung der HNF näher untersuchen zu können. Aufbauend auf der Kenntnis der nicht–modularen Algorithmen zur Berechnung der HNF ist es sehr leicht, modulare Algorithmen zu konstruieren, die die gleiche Vorgehensweise verfolgen aber zusätzlich die Kenntnis eines Vielfachen der Gitterdeterminanten zur Kontrolle der Größe der Zwischeneinträge benutzen.

Um dies zu erreichen, werden in die jeweiligen Kernalgorithmen zum einen zusätzliche Operationen integriert, die mittels Modulo–Reduktion die Größe der Einträge beschränken. Zum anderen sind zusätzliche Modulalgorithmen von Nöten, die den durch die Moduloreduktion begangenen Fehler korrigieren, indem sie die einzelnen Elemente der konkatenierten, mit α parametrisierten Einheitsmatrix eliminieren.

Analog zur Darstellung der nicht–modularen Algorithmen stellen wir zur Veranschaulichung die modularen Algorithmen für den Fall der Berechnung der HNF von ganzzahligen Matrizen mit vollem Zeilenrang dar und wiederum unterscheiden wir in zweiter Stufe drei Algorithmenklassen:

1. Zeilenweise Berechnung der HNF
2. Spaltenweise Berechnung der HNF
3. Hauptminorenweise Berechnung der HNF

In den folgenden Abschnitten stellen wir die einzelnen Algorithmenklassen vor. Dabei verzichten wir auf eine detaillierte Laufzeitanalyse aus folgender Begründung heraus.

Laufzeitverhalten der modularen Algorithmen zur HNF–Berechnung:

Die Laufzeit der modularen HNF–Algorithmen hängt in erster Linie von der Größe des verwendeten Vielfachen der Gitterdeterminanten ab. Erst in zweiter Linie spielt die Eliminationstrategie eine Rolle. Je schneller die Größe der entstehenden Zwischeneinträge die Größe des verwendeten Vielfachen der Gitterdeterminanten übersteigt, desto früher und desto öfter müssen Moduloreduktionen zur Kontrolle der Eintragsgröße durchgeführt werden.

Somit verhalten sich die modularen HNF–Algorithmen prinzipiell wie die nicht modularen HNF–Algorithmen. Bedingt durch die Größenkontrolle mittels Moduloreduktion wird eine Überschreitung des zur Verfügung stehenden Hauptspeichers ausgeschlossen. Dies geht auf Kosten der Laufzeit.

11.2. Bemerkung
Durch heuristische Erweiterungen in der Art, daß nicht nach jeder Stufe der HNF–Berechnung eine Moduloreduktion durchgeführt wird, sondern nach einer gewissen, heuristisch bestimmten Anzahl von Stufen, kann eine weitere Verbesserung der Laufzeit erreicht werden.

11.1 Zeilenweise Berechnung

Die modularen HNF–Algorithmen, die sich der zeilenweisen Elimination bedienen, basieren auf den folgenden beiden Kernalgorithmen, die sich aus den entsprechenden nicht–modularen HNF–Algorithmen herleiten.

Kernalgorithmen

11.3. Algorithmus

Modularer Kernalgorithmus zur zeilenweisen HNF–Berechnung mit Zeilennormalisierung: HNFmod_{Z1}

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, $\mathrm{rank}_{\mathbb{Z}}(A) = m$,
Vielfaches der Gitterdeterminante α

AUSGABE: $\mathrm{HNF}(A)$

[Initialisierung]
(1) $i = m-1; j = n-1; l = 0;$
(2) **while** $(\mathrm{i} \geq 0 \wedge \mathrm{j} \geq 0)$ **do**
[Zeilenweise Elimination]
(3) $A' = \begin{pmatrix} a_{0,0} & \dots & a_{0,j} \\ \vdots & \ddots & \vdots \\ a_{i,0} & \dots & a_{i,j} \end{pmatrix};$
(4) $T = \mathbf{mgcd}(\mathbf{A}');$ $\quad //a'_{i,*} \cdot T = (0, \dots, 0, \gcd(a'_{i,*}))$
(5) $A = A \cdot \begin{pmatrix} T & 0 \\ 0 & I_l \end{pmatrix} \bmod(\alpha);$
[Korrektur des Diagonaleintrags]
(6) $\alpha = \mathbf{correct\_row}(\mathbf{A}, \mathbf{i}, \alpha);$

[Zeilenweise Normalisierung]
(7) $A = \mathbf{normalize\_row}(\mathbf{A}, \mathbf{i}, \mathbf{j})$;
(8) i– –; j– –; l++;
(9) **od**
(10) **return** (A);

Entsprechend sieht der modifizierte Algorithmus mit Matrixnormalisierung wie folgt aus.

11.4. Algorithmus

Modularer Kernalgorithmus zur zeilenweisen HNF–Berechnung mit Matrixnormalisierung: HNFmod_{Z2}

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$, $\mathrm{rank}_{\mathbb{Z}}(A) = m$,
Vielfaches der Gitterdeterminante α

AUSGABE: HNF(A)

[Initialisierung]
(1) $i = m-1; j = n-1; l = 0;$
(2) **while** (i $\geq 0 \wedge$ j ≥ 0) **do**
[Zeilenweise Elimination]
(3) $A' = \begin{pmatrix} a_{0,0} & \dots & a_{0,j} \\ \vdots & \ddots & \vdots \\ a_{i,0} & \dots & a_{i,j} \end{pmatrix}$;
(4) $T = \mathbf{mgcd}(\mathbf{A}')$; $\quad //a'_{i,*} \cdot T = (0, \dots, 0, \gcd(a'_{i,*}))$
(5) $A = A \cdot \begin{pmatrix} T & 0 \\ 0 & I_l \end{pmatrix} \bmod (\alpha)$;
[Korrektur des Diagonaleintrags]
(6) $\alpha = \mathbf{correct\_row}(\mathbf{A}, \mathbf{i}, \alpha)$;
(7) i– –; j– –; l++;
(8) **od**
[Matrixnormalisierung]
(9) $A = \mathbf{normalize\_matrix}(\mathbf{A}, \mathbf{i}, \mathbf{j})$;
(10) **return** (A);

Zusätzlich zu den Modulalgorithmen **mgcd** und **normalize_row**, die wir schon in den Abschnitten 10.1.1 und 10.1.2 abgehandelt haben, enthalten die vorgestellten Kernalgorithmen dieser Algorithmenklasse den Modulalgorithmus **correct_row**.

Im folgenden Abschnitt gehen wir auf verschiedene Realisierungsmöglichkeiten für diesen Modulalgorithmus ein.

11.1.1 Korrektur der Diagonaleinträge

In diesem Abschnitt befassen wir uns mit Funktionen, die gemäß Satz 2.28 in der Lage sind, einen definierten Diagonaleintrag einer Matrix zu korrigieren.

In LiDIA sind zur Zeit zwei Funktionen dieser Art integriert.

1. Aufgrund dessen, daß α ein Vielfaches der Gitterdeterminanten des von A erzeugten Gitters $L(A)$ ist , muß nur sichergestellt werden, daß das jeweils betrachtete Diagonalelement dem größten gemeinsamen Teiler aus dem Diagonalelement und α entspricht. Dies führt uns zu dem folgenden Algorithmus.

11.5. Algorithmus

Korrektur des Diagonaleintrags: $correct\_row_{normal}$

EINGABE: $A \in \mathbf{Mat}_{m,n}(\mathbb{Z})$, Zeilenindex i,
Vielfaches der Gitterdeterminante α

AUSGABE: A mit korrigierten Diagonalelement der i–ten Zeile

(1) $g = xgcd(p, q, a_{i,(m-n)+i}, \alpha)$;
(2) $a_{*,(m-n)+i} = p \cdot a_{*,(m-n)+i} \bmod \alpha$;
(3) **if** $(a_{i,(m-n)+i} = 0)$ **then**
(4) $\quad a_{i,(m-n)+i} = \alpha$;
(5) **fi**
(6) **return** (α);

2. Domich stellt in seiner Arbeit [Dom89] einen weiteren Ansatz vor, der sich positiv auf die Ausführungseffizienz auswirkt.

   Wie bereits des öfteren erwähnt, führt die Kenntnis eines kleinen Vielfachen der Gitterdeterminante zu einer höheren Ausführungseffizienz. Im Normalfall jedoch ist das Vielfache, welches zur Verfügung steht, weit von der realen Gitterdeterminanten entfernt. Unser Bestreben sollte sich demnach darauf richten, während der Berechnung der HNF erhaltene Informationen zur weiteren Einschränkung der Größe der entstehenden Einträge zu nutzen.

   In jeder Iteration der obigen Kernalgorithmen erhalten wir eine Zeile bzw. ein Diagonalelement der HNF der erweiterten Matrix. Nach Elimination des entsprechenden Eintrags der konkatenierten, parametrisierten Einheitsmatrix, ist der verbleibende Diagonaleintrag der wirkliche Diagonaleintrag der HNF und somit ein Teiler der Gitterdeterminanten. Für die weitere Berechnung können wir nun den Eintrag α aktualisieren, indem wir den erhaltenen Diagonaleintrag aus dem Vielfachen der Gitterdeterminanten herausdividieren. Diese Vorgehensweise bezeichnet Domich als **decreasing modulus approach (DMA)**. Dies ergibt den folgenden Modulalgorithmus.

11.6. Algorithmus

Korrektur des Diagonaleintrags mit decreasing modulus approach: $correct\_row_{DMA}$

EINGABE: $A \in \mathbf{Mat}_{m,n}(\mathbb{Z})$, Zeilenindex i,
Vielfaches der Gitterdeterminante α

AUSGABE: A mit korrigierten Diagonalelement der i–ten Zeile

(1) $g = xgcd(p, q, a_{i,(m-n)+i}, \alpha)$;
(2) $a_{*,(m-n)+i} = p \cdot a_{*,(m-n)+i} \bmod \alpha$;
(3) **if** $(a_{i,(m-n)+i} = 0)$ **then**
(4) $\quad a_{i,(m-n)+i} = \alpha$;
(5) **fi**
(6) $\alpha = \frac{\alpha}{a_{i,(m-n)+i}}$;
(7) **return** (α);

11.7. Bemerkung
Der Einsatz des *decreasing modulus approach* führt zur Verbesserung der Gesamtlaufzeit. Der Grad der Verbesserung hängt direkt von der Verteilung der Einträge auf der Diagonalen ab.

11.1.2 Algorithmenübersicht

In der aktuellen Implementierung von LiDIA sind, wie wir bereits im Abschnitt 10.1.3 dargelegt haben, 22 mgcd–Modulalgorithmen und 5 Normalisierungsalgorithmen implementiert. Dazu kommen nun 2 Korrekturalgorithmen (Tabelle 11.1).

Kennzahl	Korrekturalgorithmus	Bezeichnung
0	$correct\_row_{normal}$	normale Korrektur
1	$correct\_row_{DMA}$	Decreasing Modulus Approach

Tabelle 11.1: Zusammenstellung Korrekturalgorithmen

Somit ergeben sich kombinatorisch betrachtet für den Kernalgorithmus $HNFmod_{Z1}$ 44 und für den Kernalgorithmus $HNFmod_{Z2}$ 220 verschiedene Algorithmen zur Berechnung der HNF.

11.2 Spaltenweise Berechnung der HNF

Analog zu der im vorangegangenen Abschnitt angewendeten Vorgehensweise lassen sich modulare Algorithmen konstruieren, die sich der spaltenweisen Eliminationsstrategie be-

dienen. Wir ergänzen wiederum an geeigneten Stellen den Kernalgorithmus um Reduktionsschritte. Wir erhalten somit den folgenden Kernalgorithmus:

11.8. Algorithmus

Modularer Kernalgorithmus zur spaltenweisen HNF–Berechnung: HNFmod_{GLS}

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ mit $\text{rank}_{\mathbb{Z}}(A) = m$,
Vielfaches der Gitterdeterminante α.
AUSGABE: $\text{HNF}(A)$.

[Bestimmung einer regulären Teilmatrix]
(1) $v = \mathbf{lininc}_{\mathbb{Z}}(\mathbf{A})$;
(2) $T = regular\_sort(v)$;
(3) $A = A \cdot T$; // $A \cdot T = (C|E)$
(4) $A.divide\_h(C, E)$; // $det_{\mathbb{Z}}(E) \neq 0$
(5) $A = \mathbf{update\_relations}(\mathbf{A}, \mathbf{T}) \bmod (\alpha)$;
[HNF–Berechnung Phase 1]
(6) **for** $(i = 0; i < n - m; i++)$ **do**
(7) $b = a_{*,i}$;
(8) $(x, k) = \mathbf{solve}_{\mathbb{Z}}(\mathbf{E}, \mathbf{b})$; // $E \cdot x = k \cdot b$
(9) $y = (0, \ldots, 0, -k, 0, \ldots, 0, x)$;
(10) $y = \frac{y}{\gcd(c)}$;
(11) $A = \mathbf{update\_relations}(\mathbf{A}, \mathbf{basis\_completion}(\mathbf{y})) \bmod (\alpha)$;
(12) **od**
[HNF–Berechnung Phase 2]
(13) **for** $(i = 0; i < n; i++)$ **do**
(14) $b = \begin{pmatrix} e_{i+1,i} \\ \vdots \\ e_{n-1,i} \end{pmatrix}$;
(15) $E' = \begin{pmatrix} e_{i+1,i+1} & \cdots & e_{i+1,n-1} \\ \vdots & \ddots & \vdots \\ e_{n-1,i+1}) & \cdots & e_{n-1,n-1} \end{pmatrix}$;
(16) $(x, k) = \mathbf{solve}_{\mathbb{Z}}(\mathbf{E}', \mathbf{b})$; // $E' \cdot x = k \cdot b$
(17) $y = (0, \ldots, 0, -k, 0, \ldots, 0, x)$;
(18) $y = \frac{y}{\gcd(x)}$;
(19) $A = \mathbf{update\_relations}(\mathbf{A}, \mathbf{basis\_completion}(\mathbf{y})) \bmod (\alpha)$;
(20) **od**
[Korrektur der Diagonaleinträge]
(21) **for** $(i = m - 1; i \geq 0; i--)$ **do**
(22) $\alpha = \mathbf{correct\_row}(\mathbf{A}, \mathbf{i}, \alpha)$;
(23) **od**
(24) **return** (A);

Auf die verschiedenen in LiDIA implementierten Möglichkeiten, die Modulalgorithmen *lininc*, *solve*, *basis_completion* und *update_relations* zu realisieren, sind wir bereits im Detail in entsprechenden Abschnitten im Kapitel *Nichtmodulare HNF–Algorithmen* eingegangen. Somit sind bereits alle Algorithmen dieser Algorithmenklasse vollständig beschrieben.

11.2.1 Algorithmenübersicht

Die Klasse der nichtmodularen HNF–Algorithmen, die sich der spaltenweisen Eliminationsstrategie bedienen, umfaßt gemäß der Betrachtungen im Abschnitt 10.2.5 128 Algorithmen. Kombiniert mit den zwei Möglichkeiten, die durch die Moduloreduktion eingebrachten Fehler zu korrigieren (siehe Abschnitt 11.1), ergeben sich für diese Algorithmenklasse insgesamt 256 Algorithmen.

11.3 Hauptminorenweise Berechnung der HNF

Zur Vervollständigung unserer Darstellung der modularen HNF–Algorithmen beschreiben wir in diesem Abschnitt die Klasse von Algorithmen, die auf der Vorgehensweise von Kannan und Bachem [KB79] beruhen.

Wiederum wird der Kernalgorithmus der nichtmodularen HNF–Algorithmen, die sich der Vorgehensweise von Kannan und Bachem bedienen, durch die Integration von Moduloreduktions- und Korrekturphasen in einen modulare HNF–Kernalgorithmus verwandelt.

Wir erhalten den folgenden Kernalgorithmus:

11.9. Algorithmus

Modularer Kernalgorithmus nach Kannan und Bachem:
$\text{HNFmod}_{KannanBachem}$

EINGABE: $A \in \mathbf{Mat}_{m\times n}(\mathbb{Z})$ mit $\text{rank}_{\mathbb{Z}}(A) = m$,
Vielfaches der Gitterdeterminante α.
AUSGABE: $\text{HNF}(A)$

[Initialisierung]
(1) $T = I_n$
(2) $l = 2$;
[HNF–Berechnung]
(3) **while** $(l < n)$ **do**
(4) **for** $(j = 1; j < l; j++)$ **do**
(5) $p = n - j$;

[Handling für reguläre Teilmatrix]
(6) **if** $(a_{p,p} = 0)$ **then**
(7) **for** (i = p; i $\geq 0 \wedge a_{p,i} = 0$; i– –) **do**
(8) **od**
(9) $A.swap\_columns(i, p)$;
(10) **fi**
[Elimination]
(11) $g = xgcd(p, q, a_{p,n-l}, a_{p,p})$;
(12) $U = \begin{pmatrix} -\frac{a_{p,p}}{g} & p \\ \frac{a_{p,n-l}}{g} & q \end{pmatrix}$;
(13) $(a_{*,n-l}, a_{*,p}) = (a_{*,n-l}, a_{*,p}) \cdot U \bmod (\alpha)$;
(14) **od**
[Normalisierung]
(15) **A**.**normalize_matrix**(**p**, **p**);
(16) **od**
(17) **return** (A;);

11.3.1 Algorithmenübersicht

Die hier vorgestellt Algorithmenklasse umfaßt insgesamt 8 Algorithmen. Die nichtmodulare Algorithmenklasse, die sich der Vorgehensweise von Kannan und Bachem bedient, umfaßt 4 Algorithmen, die mit 2 Algorithmen zur Korrektur der Diagonaleinträge kombiniert werden.

Teil V

Framework III: Vorgehensweise und Anwendungsbeispiel

Kapitel 12

Vorgehensweise

Um das in den ersten Teilen dieser Arbeit vorgestellte Framework auf eine spezielle Klasse von Eingabematrizen anzuwenden, ist es sinnvoll, wie folgt vorzugehen.

12.1 Vorbereitungsphase

In der Vorbereitungsphase schaffen wir die Grundlagen für die nachfolgende Konstruktionsphase.

1. **Analyse der Matrixklasse**
   Zuerst analysiert man die Matrizen der Eingabematrixklasse in Bezug auf die Eintragsdichte, die maximale Eintragsgröße, die durchschnittliche Eintragsgröße und die Verteilung der Einträge.
2. **Auswahl charakteristischer Matrizen**
   Dann wählt man typische, charakteristische Beispielmatrizen aus der Menge der Eingabematrizen aus, d.h. man wählt Matrizen aus, die in Bezug auf die oben analysierten Parameter dem Durchschnitt entsprechen.

12.2 Iterative Konstruktionsphase

Mit Hilfe der ausgewählten Matrizen konstruiert man einen Hybrid–HNF–Algorithmus, indem man in einem iterativen Prozeß, die folgenden Phasen durchläuft:

1. **Analyse der (Rest-)Matrizen**
   In dieser Phase analysiert man die ausgewählten (Rest-)Matrizen in Bezug auf die Eintragsdichte, die maximale Eintragsgröße, die durchschnittliche Eintragsgröße und die Verteilung der Einträge. Desweiteren legt man die Strategie für den kommenden Auswahlprozeß fest. In welchem Verhältnis steht die Effizienz zur Kontrolle der Größe

der Zwischenergebnisse? Ist die Kontrolle der Eintragsdichte höher zu bewerten als die Kontrolle des verwendeten Speicherplatzes?

2. **Auswahl eines geeigneten HNF–Algorithmus**
   Anhand der ermittelten Parameter wählen wir einen HNF–Algorithmus aus der Menge aller zur Verfügung stehenden, implementierten HNF–Algorithmen aus, der unter Einhaltung der Rahmenbedingungen und unter Berücksichtigung der gewählten Strategie maximale Effizienz besitzt. Dazu verwenden wir zum einen die im Rahmen des Aufbaus des Frameworks gesammelten Laufzeitinformationen. Zum anderen verwenden wir Informationen, die wir gewinnen, indem wir HNF–Berechnungen auf den ausgewählten (Rest-)Matrizen mit den implementierten HNF–Algorithmen ausführen. Desweiteren legen wir unter Berücksichtigung der Eintragsdichte der (Rest-)Matrizen eine Matrixrepräsentation und unter Berücksichtigung der Größe des maximalen Eintrags der (Rest-)Matrizen den zugrundeliegenden Datentyp fest.

3. **Festlegung eines Abbruch–Kriteriums**
   Anschließend legen wir unter Berücksichtigung der im vorangegangen Schritt gesammelten Laufzeitinformationen fest, wie lange dieser HNF–Algorithmus rechnen darf, bevor in eine neue Iteration gewechselt wird. D.h. wir legen ein Abbruchkriterium für den ausgewählten HNF–Algorithmus und damit für die aktuelle Stufe des Hybrid–HNF–Algorithmus fest.

   12.1. Bemerkung
   Im Extremfall wird nach jedem HNF–Berechnungschritt, d.h. nach jeder Zeileneliminination oder nach jeder Spaltenelimination, etc., eine neue Phase eingeleitet. Damit würde sich der Hybrid–HNF–Algorithmus sehr dicht an die Veränderungen der noch zu verarbeitenden Restmatrizen anpassen.

In jedem Durchlauf erhalten wir eine neue Phase des gewünschten Hybrid–HNF–Algorithmus. Die Anzahl der Durchläufe entspricht somit der Anzahl der Phasen des HNF–Algorithmus. Die Iteration endet, wenn die HNF für alle ausgewählten Matrizen berechnet ist.

In folgenden Kapitel zeigen wir anhand eines Anwendungsbeispiels konkret, was in jedem Schritt zu tun ist, welche Ergebnisse erzielt werden und wie der Hybrid–HNF–Algorithmus schließlich aussieht.

Kapitel 13

Anwendungsbeispiel: Klassengruppenberechnung

Die Public–Key–Kryptographie hat sich in den letzten Jahren zu einem wichtigen Baustein in modernen Sicherheitsarchitekturen entwickelt. Sie bildet das kryptographische Rückgrad, welches es ermöglicht, auf einfache Art und Weise sichere Verbindungen aufzubauen, elektronische Nachrichten zu signieren, Zugangsschutz zu realisieren, usw. Die Sicherheit des verwendeten Public–Key–Verfahrens ist dabei von besonderer Bedeutung [BM99]. Die meisten in der Praxis verwendeten Public–Key–Systeme beziehen ihre Sicherheit aus der Schwierigkeit, natürliche Zahlen in ihre Primfaktoren zu zerlegen (RSA–Verfahren) oder diskrete Logarithmen in endlichen Primkörpern zu berechnen (Digital Signature Algorithm (DSA)). Beides gilt heute bei geeigneter Wahl der Parameter als sehr schwierig, obwohl bisher nicht bewiesen werden konnte, daß diese Probleme wirklich schwierig sind. Es liegt somit im Bereich des möglichen, daß früher oder später ein Wissenschaftler einen effizienten Algorithmus findet, der sehr schnell natürliche Zahlen faktorisieren bzw. diskrete Logarithmen berechnen kann. Es ist somit nötig, neue Probleme zu identifizieren, die sich als Grundlage für die Entwicklung und Implementierung von alternativen Public–Key–Systemen eignen. Eine solche Alternative ist die Berechnung von diskreten Logarithmen in der Klassengruppe eines algebraischen Zahlkörpers. Um die Sicherheit dieser Alternative beurteilen zu können, ist es notwendig, die Schwierigkeit dieses Problems sowohl theoretisch als auch praktisch zu untersuchen.

Die vorliegende Arbeit steht unter anderem auch im Kontext der Arbeit [Jr.99] von Michael Jacobson Jr. und der Arbeit [Nei01] von Stefan Neis. Das übergeordnete Ziel dieser drei Arbeiten ist es, die Schwierigkeit der Berechnung diskreter Logarithmen in der Klassengruppe eines algebraischen Zahlkörpers experimentell zu untersuchen. Michael Jacobson Jr. stellt dazu in seiner Arbeit [Jr.99] einen neuen Algorithmus zur Berechnung der Klassengruppe und zur Berechnung von diskreten Logarithmen in quadratischen Zahlkörpern vor, während Stefan Neis in seiner Arbeit sich den gleichen Problemstellungen in Zahlkörpern höherer Dimension widmet. Gemeinsam ist ihren Algorithmen, daß sie eine Methode benötigen, um große, dünnbesetzte, ganzzahlige Relationenmatrizen auf Hermite–Normalform zu transformieren.

In diesem Teil der vorliegenden Arbeit wird das in den vorangegangenen Teilen dieser

Arbeit vorgestellte Framework auf den speziellen Kontext der Klassengruppenberechnung angewendet. Genauer gesagt, wir sind daran interessiert, Hybrid–Algorithmen im Sinne unserer Lösungsidee zu entwickeln, die in der Lage sind, in einer vorgegebenen Hardware- und Softwareumgebung (Relationen-)Matrizen, die als Zwischenergebnis moderner Algorithmen zur Berechnung der Klassengruppe algebraischer Zahlkörper entstehen, auf HNF zu bringen.

Wir unterscheiden somit die folgenden beiden Klassen von Eingabematrizen:

1. **Relationenmatrizen nach Jacobson**
   Die erste Klasse, die wir im folgenden mit $ClassGroup_{Jacobson}$ bezeichnen, umfaßt Relationenmatrizen, die als Zwischenergebnis bei der Berechnungen der Klassengruppe mittels des von Michael Jacobson Jr. entwickelten Algorithmus entstehen. Auf Details des Algorithmus zur Erzeugung der Relationenmatrizen gehen wir an dieser Stelle nicht ein, sondern verweisen auf die Arbeit von Michael Jacobson Jr. [Jr.99].

2. **Relationenmatrizen nach Neis**
   Die zweite Klasse von Matrizen trägt den Namen $ClassGroup_{Neis}$ und beinhaltet die Relationenmatrizen, die der Algorithmus von Stefan Neis zur Berechnung der Klassengruppe als Zwischenergebnis erzeugt. Eine detaillierte Beschreibung des Algorithmus von Stefan Neis findet sich in der Arbeit [Nei01].

Die Hardware- bzw. Softwareumgebung setzt sich wie folgt zusammen:

Maschine	Sparc Ultra 300 MHz
Hauptspeicher	256 MB RAM
Swap	256 MB
Betriebssystem	SUN Solaris 2.5
C++ Compiler	GNU g++ 2.8.1

In den folgenden Abschnitten gehen wir im Detail auf die einzelnen Schritte der im vorangegangenen Kapitel beschriebenen Vorgehensweise ein. Wir erläutern, was in jedem Schritt zu tun ist und welche Ergebnisse erzielt werden. Parallel zeigen wir dies exemplarisch für die Eingabematrixklassen $ClassGroup_{Jacobson}$ und $ClassGroup_{Neis}$.

13.1 Vorbereitungsphase

13.1.1 Analyse der Matrixklasse

Das Ziel dieses ersten Schritts besteht darin, statistische Informationen über die Matrizen der betrachteten Eingabematrixklasse zu sammeln. Dazu untersuchen wir die Matrizen der Eingabematrixklasse im Hinblick auf

1. das Gewicht,

2. die Verteilung der Nicht–Nulleinträge,
3. die durchschnittliche Eintragsgröße,
4. die Größenverteilung der Einträge und
5. die maximale Eintragsgröße.

Mittels dieser Informationen sind wir später in der Lage, die richtigen HNF–Referenzdaten zur Auswahl eines geeigneten HNF–Algorithmus heranzuziehen.

Konkret auf das Umfeld der Klassengruppenberechnung angewendet ergeben sich die in den folgenden Abschnitten dargestellten Ergebnisse.

13.1.1.1 $ClassGroup_{Jacobson}$

Wie bereits dargelegt umfaßt die Eingabematrixklasse $ClassGroup_{Jacobson}$ Relationenmatrizen, die als Zwischenergebnis bei der Berechnungen der Klassengruppe mittels des von Michael Jacobson Jr. entwickelten Algorithmus entstehen. Alle Untersuchungen, die wir in dieser Arbeit bzgl. dieser Matrixklasse durchführen, basieren auf empirischen Daten, die wir aus einer Datenbasis von über 350 Relationenmatrizen unterschiedlicher Dimensionen im Laufe der letzten Jahre ermittelt haben.

1. **Eintragsdichte:**
   Untersuchen wir zunächst die durchschnittliche Eintragsdichte einer Relationenmatrix der Klasse $ClassGroup_{Jacobson}$. Dazu haben wir die Eintragsdichte aller uns vorliegenden Relationenmatrizen berechnet und gemäß des prozentualen Anteils an der Gesamtanzahl der Matrizen in der Abbildung 13.1 zusammengetragen.

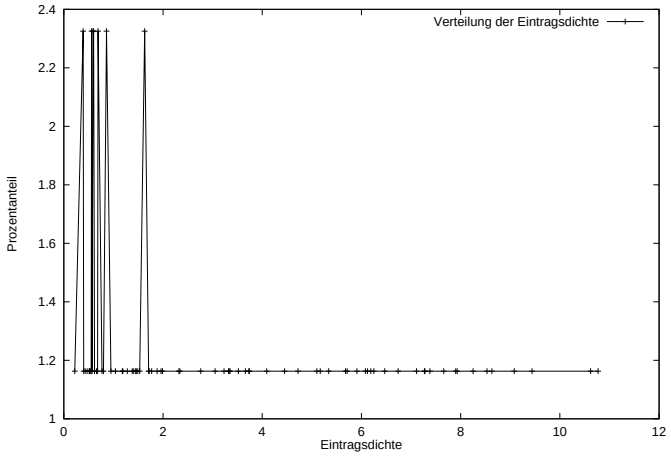


Abbildung 13.1: Prozentuale Eintragsdichteverteilung $ClassGroup_{Jacobson}$

Die absolute Eintragsdichte bewegt sich zwischen $0,2\%$ und $10,8\%$. Auffallend ist die Häufung bei einer Dichte von $0,6\%$ mit einem Prozentanteil von ungefähr $10,6\%$.

Die durchschnittliche Eintragsdichte von Relationenmatrizen des Typs $ClassGroup_{Jacobson}$ liegt bei $3,307\%$.

2. **Verteilung der Einträge:**
In diesem Abschnitt untersuchen wir die Verteilung der Einträge. Zu diesem Zweck ermitteln wir die durchschnittliche Verteilungsfunktion über die Gesamtheit der zur Verfügung stehenden Relationenmatrizen.

Die Abbildung 13.2 zeigt die ermittelte durchschnittliche Spaltenverteilungsfunktion. Dies bedeutet, wir unterteilen die Spalten einer Matrix in 100 gleichgroße Blöcke und ermitteln, wieviel Prozent der Nicht–Nulleinträge der jeweiligen Matrix in dem jeweils betrachteten Block liegen. Anschließend mitteln wir über alle uns zur Verfügung stehenden Relationenmatrizen.

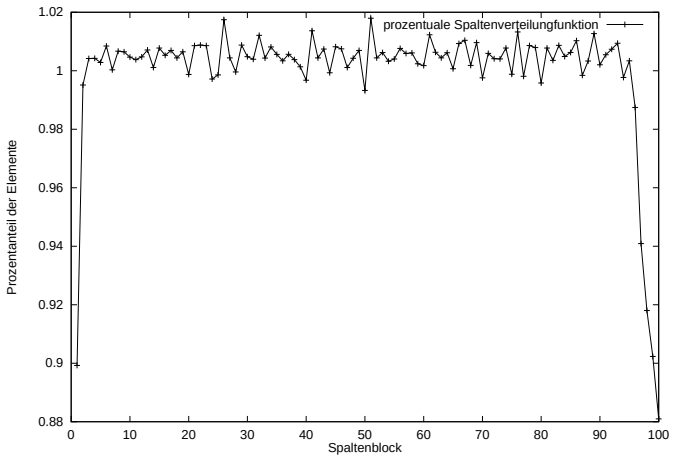


Abbildung 13.2: Prozentuale Spaltenverteilungsfunktion $ClassGroup_{Jacobson}$

Es wird deutlich, daß die Anzahl der Einträge pro Block und somit implizit pro Spalte (Relation) im Durchschnitt fast konstant ist, mit Ausnahme der ersten und der letzten Relationen der Relationenmatrizen.

Untersuchen wir nun analog dazu die durchschnittliche zeilenweise Verteilung der Einträge (Abbildung 13.3).

Man erkennt sehr deutlich, daß die Anzahl der Nicht–Nulleinträge mit zunehmendem Zeilenindex abnimmt. In den ersten 10 der Zeilenblöcke befinden sich über 80% der Nicht–Nulleinträge.

3. **Größe der Einträge:**
Neben der Verteilung der Nicht–Nulleinträge interessieren wir uns für die Größe der Einträge. Über alle Relationenmatrizen hinweg gesehen liegen die Einträge im Intervall $[-19, 19]$.

Die Größenverteilung je Relationenmatrix ähnelt dabei einer gestreckten Gaußverteilung um den Nullpunkt.

Die durchschnittliche Eintragsgröße der Absolutbeträge aller Nicht–Nulleinträge über alle uns zur Verfügung stehenden Relationenmatrizen beträgt $1,07$.

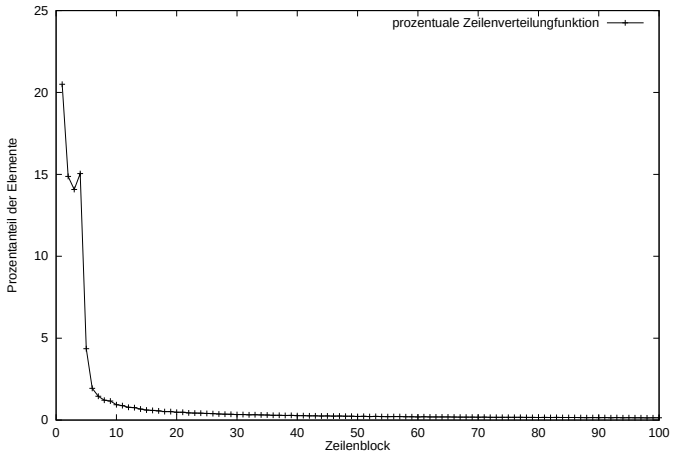


Abbildung 13.3: Prozentuale Zeilenverteilungsfunktion $ClassGroup_{Jacobson}$

13.1.1.2 $ClassGroup_{Neis}$

Die zweite Klasse von Matrizen trägt $ClassGroup_{Neis}$ beinhaltet die Relationenmatrizen, die der Algorithmus von Stefan Neis zur Berechnung der Klassengruppe als Zwischenergebnis erzeugt. Die Datenbasis, auf der wir die empirischen Daten dieser Matrixklasse ermitteln, umfaßt 249 Relationenmatrizen unterschiedlicher Dimensionen.

1. **Eintragsdichte:**
   Zur Analyse der Eintragsdichte der Relationenmatrizen dieser Klasse untersuchen wir zunächst, analog der Vorgehensweise für die Matrizen der Matrixklasse $ClassGroup_{Jacobson}$, die prozentuale Verteilung des Eintragsdichte bezogen auf die Gesamtheit der unserer Betrachtung zugrundeliegenden Relationenmatrizen (Abbildung 13.4)

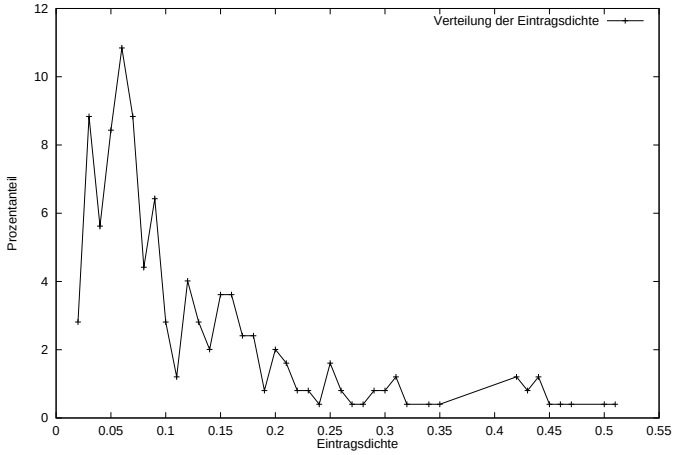


Abbildung 13.4: Prozentuale Eintragsdichteverteilung $ClassGroup_{Neis}$

Man erkennt, daß

(a) die absolute Eintragsdichte sich zwischen $0,01\%$ bis $0,51\%$ bewegt und

(b) der Prozentanteil mit steigender Dichte abnimmt.

Dies führt zu einer durchschnittlichen Eintragsdichte pro Relationenmatrix von nur $0,1296\%$. Die Eintragsdichte der Matrizen der Klasse $ClassGroup_{Neis}$ ist somit wesentlich geringer als die der Matrizen der Klasse $ClassGroup_{Jacobson}$.

2. **Verteilung der Einträge:**
Untersuchen wir nun die Verteilung der Nicht–Nulleinträge. Die Abbildung 13.5 zeigt die ermittelte durchschnittlich Spaltenverteilungsfunktion.

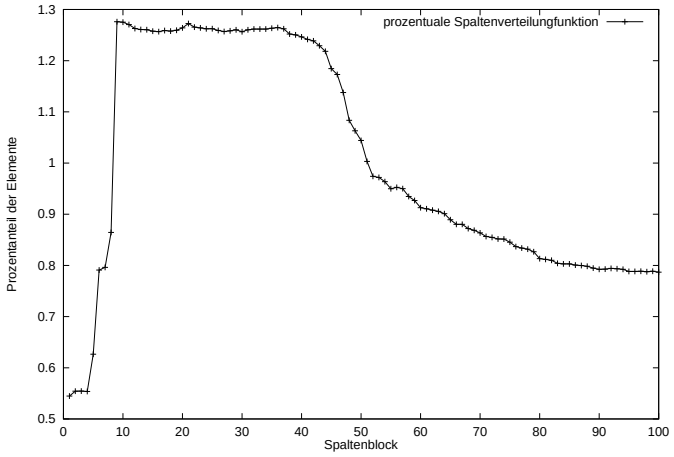


Abbildung 13.5: Prozentuale Spaltenverteilungsfunktion $ClassGroup_{Neis}$

Man kann anhand dieser Abbildung drei Bereiche unterscheiden:

(a) Der erste Bereich erstreckt sich von Block 0 bis ca. Block 8 der Spalten einer Relationenmatrix. In diesem Bereich steigt der prozentuale Anteil der Nicht–Nulleinträge von ca. $0,54\%$ auf $1,28\%$ an.

(b) Der zweite Bereich erstreckt sich von Block 8 bis Block 45 der Relationen. In diesem Bereich liegt der prozentuale Anteil fast konstant bei $1,28\%$.

(c) Im dritten Bereich, der sich von Block 45 bis zum Ende erstreckt, fällt der Prozentanteil langsam von $1,28\%$ auf ca. $0,79\%$ ab.

Eine Begründung für diese Verteilung findet sich in der Arbeit von Stefan Neis [Nei01].

Untersuchen wir nun die Zeilenverteilungsfunktion (Abbildung 13.6).

In Übereinstimmung mit den Relationenmatrizen der Klasse $ClassGroup_{Jacobson}$ liegen über 80% der Nicht–Nulleinträge in den ersten 10 Blöcken der Zeilen einer Relationenmatrix.

3. **Größe der Einträge:**
Die Einträge der Relationenmatrizen der Klasse $ClassGroup_{Neis}$ liegen absolut gesehen zwischen 0 und 8.

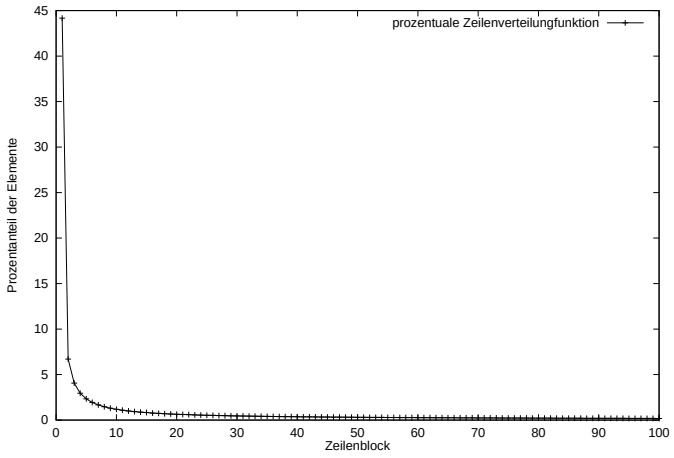


Abbildung 13.6: Prozentuale Zeilenverteilungsfunktion $ClassGroup_{Neis}$

Im Gegensatz zu den Matrizen der Klasse $ClassGroup_{Jacobson}$ sind alle Einträge größer gleich 0.

Die durchschnittliche Eintragsgröße über alle uns zur Verfügung stehenden Relationenmatrizen und über die Absolutbeträge aller Nicht–Nulleinträge beträgt 1,47.

13.1.2 Auswahl charakteristischer Matrizen

Für jeder der beiden Eingabematrixklassen $ClassGroup_{Neis}$ und $ClassGroup_{Jacobson}$ wurden 20 charakteristische Matrizen ausgewählt, die in Bezug auf die oben untersuchten Parameter im Durchschnitt liegen. Um die Darstellung der Vorgehensweise zu vereinfachen, beschränken wir uns im folgenden auf eine dieser Beispielmatrizen, die sich gut zur Illustration eignet. Die ermittelten Ergebnisse wurden auf der Basis der Gesamtmenge ermittelt.

Die Beispielmatrix der Klasse $ClassGroup_{Neis}$ bezeichnen wir mit BSP_{Neis} und analog die Beispielmatrix der Klasse $ClassGroup_{Jacobson}$ mit $BSP_{Jacobson}$.

13.2 Iterative Konstruktionsphase

Aufbauend auf den Eigenschaften der ausgewählten Matrizen, beginnen wir nun mit der Konstruktion des Hybrid–HNF–Algorithmus. Inwieweit der konstruierte Algorithmus für die gesamte Klasse von Eingabematrizen geeignet ist, hängt direkt von der Güte der ausgewählten Matrizen ab.

13.2.1 Analyse der (Rest-)Matrizen

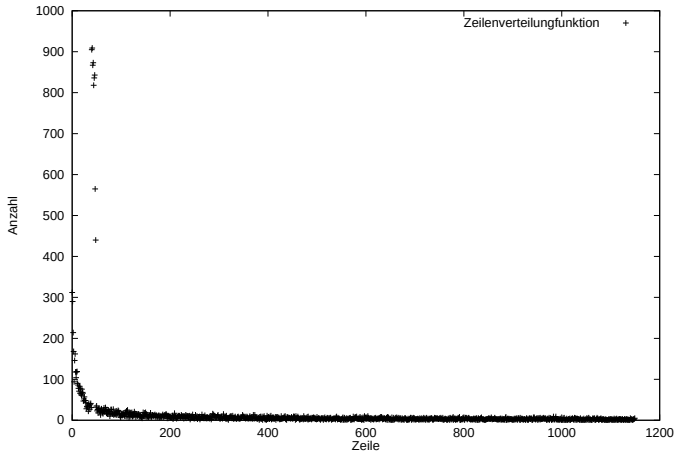
In dieser Phase untersuchen wir zunächst die Eigenschaften der ausgewählten Matrizen und legen anschließend eine Strategie für den folgenden Auswahlprozeß fest. Für die betrachteten Eingabematrixklassen sind wir zu den folgenden Ergebnissen gelangt:

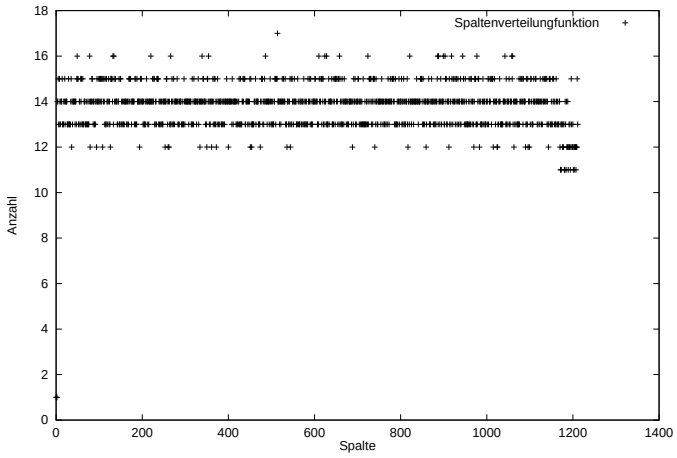
13.2.1.1 $ClassGroup_{Jacobson}$

Analyseergebnisse:

Dimension der Restmatrix	1150 x 1212
$L_0(A)$ $(\varrho(A))$	16706 (1.199 %)
$L_\infty(A)$	3
durchschnittliche Eintragsgröße	1.016
max. Elementanzahl pro Zeile	909
max. Elementanzahl pro Spalte	17

Eintragsgröße	Anzahl
-3	7
-2	128
-1	8068
1	8385
2	114
3	4





Strategie:
In der ersten Iteration, in der wir uns befinden, stehen uns noch alle Ressourcen (Hauptspeicher, etc.) in vollem Umfang zur Verfügung. Aufgrund der Beobachtungen, die wir bei der Analyse der Problemstellung gemacht haben, richtet sich unser Bestreben danach, einen HNF–Algorithmus zu finden, der erstens möglichst effizient ist und zweitens der Dünnbesetztheit Rechnung trägt und versucht, die Eintragsdichte nur geringfügig zu erhöhen.

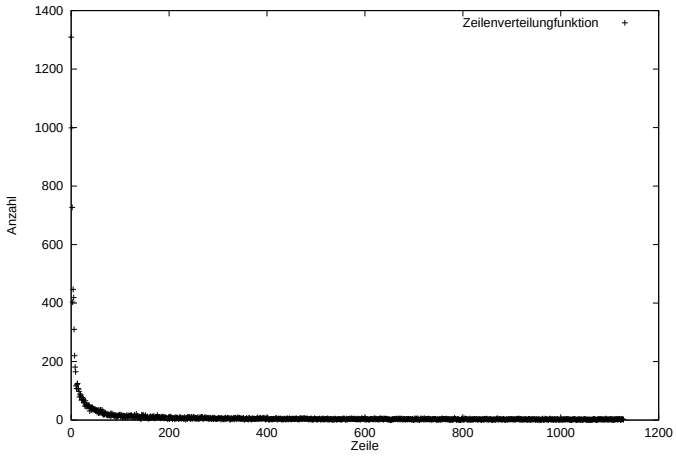
13.1. Bemerkung
In der aktuell verwendeten Vorgehensweise ist die Festlegung der weiteren Strategie eine intellektuelle, nicht automatisierte Entscheidung.

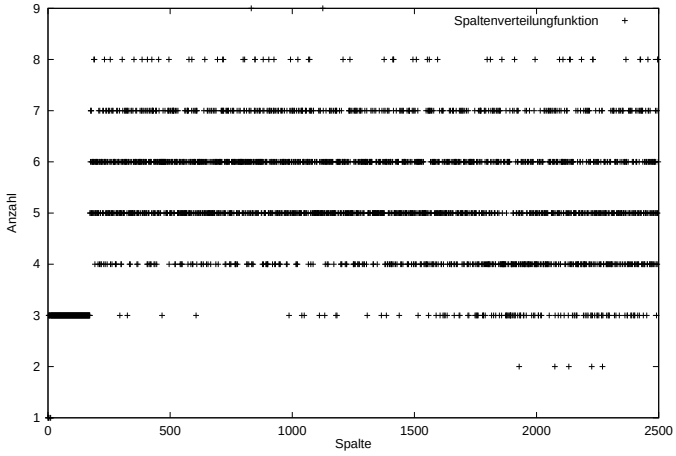
13.2.1.2 $ClassGroup_{Neis}$

Analyseergebnisse:

Dimension der Restmatrix	1129 x 2499
$L_0(A)$ $(\varrho(A))$	13155 (0.466262678676430714 %)
$L_\infty(A)$	5
durchschnittliche Eintragsgröße	1.2646902318510072216
max. Elementanzahl pro Zeile	1309
max. Elementanzahl pro Spalte	9

Eintragsgröße	Anzahl
1	9726
2	3385
3	36
4	7
5	1





Strategie:
Für die Matrix BSP_{Neis} legen wir aus den gleichen Gründen, wie oben dargestellt, die gleiche Strategie fest, d.h. man wähle einen HNF–Algorithmus, der erstens möglichst effizient ist und zweitens der Dünnbesetztheit in geeigneter Art und Weise Rechnung trägt.

13.2.2 Auswahl eines geeigneten HNF–Algorithmus

Zur Auswahl eines geeigneten HNF–Algorithmus ziehen wir zwei Informationsquellen zu Rate:

1. Zunächst führen wir HNF–Berechnungen auf den ausgewählten Eingabematrizen und protokollieren das Laufzeitverhalten, d.h. während der Durchführung der HNF–Berechnung wird
   (a) die Entwicklung des maximalen Eintrags,
   (b) die Entwicklung der Eintragsdichte und
   (c) die Laufzeit pro Iteration

beobachtet. Bei einer Überschreitung vorgegebener Schranken beenden wir jeweils die Berechnung. Die ermittelten Laufzeitdaten sind für die Auswahl eines geeigneten HNF–Algorithmus von vorrangiger Bedeutung.

Die für die Matrizen $BSP_{Jacobson}$ und BSP_{Neis} ermittelten Laufzeitdaten sind in den Anhängen B und C dargestellt.

2. Als zweite Informationsquelle berücksichtigen wir die ermittelten und im vorangegangenen Teil dieser Arbeit dargestellten Frameworkdaten. Diese Daten dienen uns lediglich als weiterer Hinweisgeber für die Auswahl.

Aufgrund der getroffenen Strategieentscheidung und der ermittelten Daten sind die modularen HNF–Algorithmen und die HNF–Algorithmen, die sich einer spaltenweisen Eliminationsstrategie bedienen, für die beiden betrachteten Matrixklassen ungeeignet. Der Rechenaufwand zur Berechnung eines Vielfachen der Gitterdeterminanten bzw. zum Lösen eines linearen Gleichungssystem ist im Vergleich zu der Laufzeit der restlichen, nicht modularen HNF–Algorithmen einfach zu hoch.

Aus den verbleibenden HNF–Algorithmen haben wir uns im Hinblick auf die beiden betrachteten Matrixklassen für einen HNF–Algorithmus entschieden, der sich der zeilenweisen Eliminationsstrategie bedient und sich aus dem HNF–Kernalgorithmus HNF_{Z2} und den Modulalgorithmen $mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 1)$ und $Normalize_{ChouCollins}$ zusammensetzt. Wie man anhand der Laufzeitergebnisse der Anhänge B und C sehen kann, ist bei diesem Algorithmus das Verhältnis zwischen Erhaltung der Dünnbesetztheit und Effizienz der Berechnung gut. Desweiteren weisen die ermittelten Kennwerte unseres Frameworks ebenfalls auf eine Eignung hin.

Aufgrund der geringen Eintragsdichte, den damit verbundenen Effizienzvorteilen der spaltenweisen Datenablage, und der geringen Eintragsgröße, wird dem ausgewählten HNF–Algorithmus als Matrixrepräsentation die spaltenorientierte, dünnbesetzte Ablage über dem Datentyp *long* zugrundegelegt.

13.2. Bemerkung
Die Auswahl eines geeigneten HNF–Algorithmus ist in der aktuellen Vorgehensweise eine intellektuelle Entscheidung, die zu einem großen Teil auf Erfahrung basiert. In der Regel ist eine eindeutige Auswahl eines Algorithmus nicht direkt möglich. Mehrere Algorithmen erscheinen aufgrund der zur Verfügung stehenden Datenlage geeignet.

13.2.3 Festlegung eines Abbruchkriteriums

Zur Festlegung eines geeigneten Abbruchkriteriums ziehen wir die Laufzeitergebnisse des im vorangegangen Schritt ausgewählten HNF–Algorithmus angewendet auf $BSP_{Jacobson}$ und BSP_{Neis} heran.

In den Abbildungen 13.7, 13.8 und 13.9 sind die Laufzeitergebnisse für die Matrix $BSP_{Jacobson}$ zu sehen. Die Abbildungen 13.10, 13.11 und 13.12 zeigen die ermittelten Daten für die Matrix BSP_{Neis}.

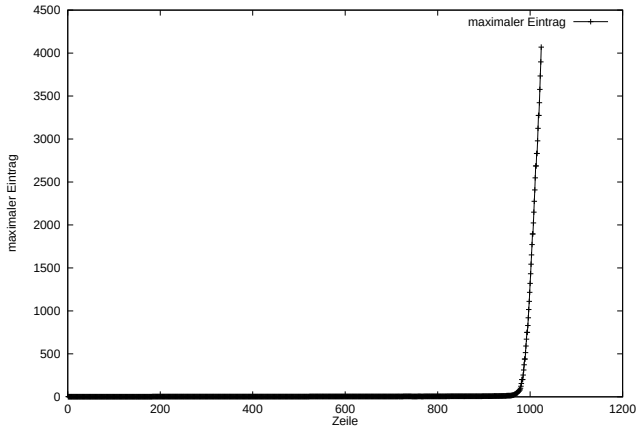


Abbildung 13.7: $BSP_{Jacobson}$: Entwicklung maximaler Eintrag

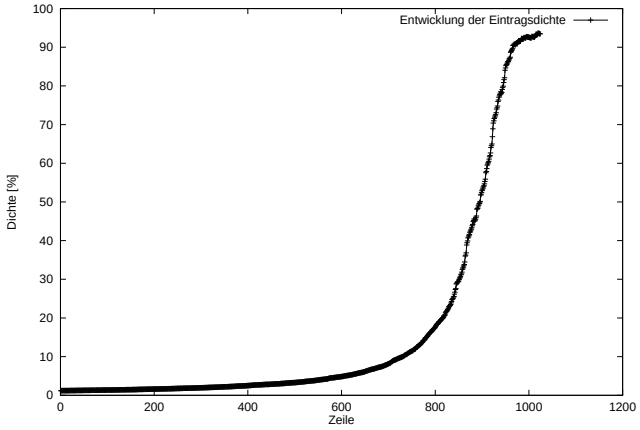


Abbildung 13.8: $BSP_{Jacobson}$: Entwicklung Eintragsdichte

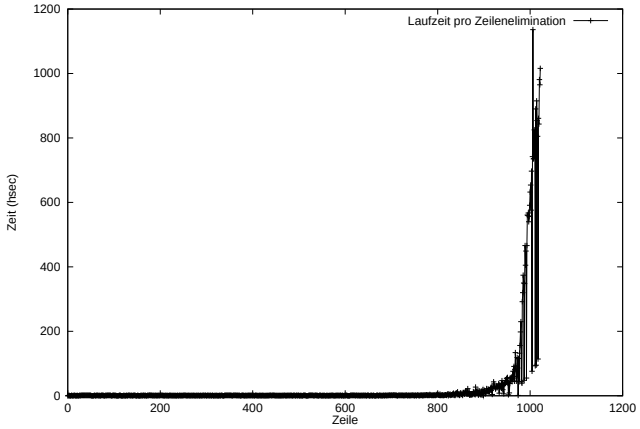


Abbildung 13.9: $BSP_{Jacobson}$: Laufzeit pro Iteration

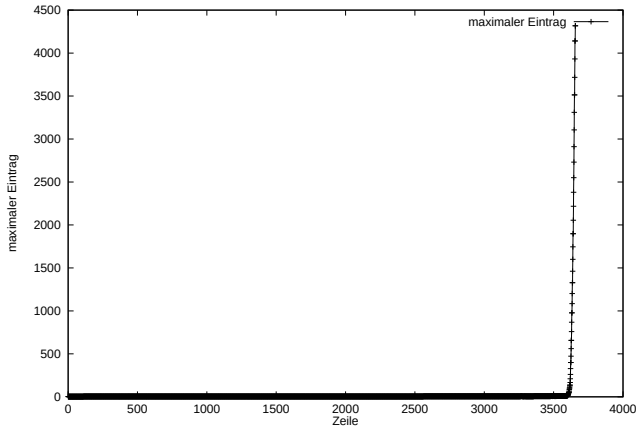


Abbildung 13.10: BSP_{Neis}: Entwicklung maximaler Eintrag

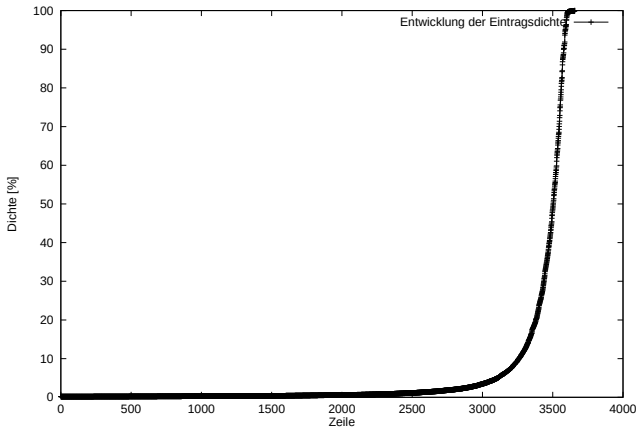


Abbildung 13.11: BSP_{Neis}: Entwicklung Eintragsdichte

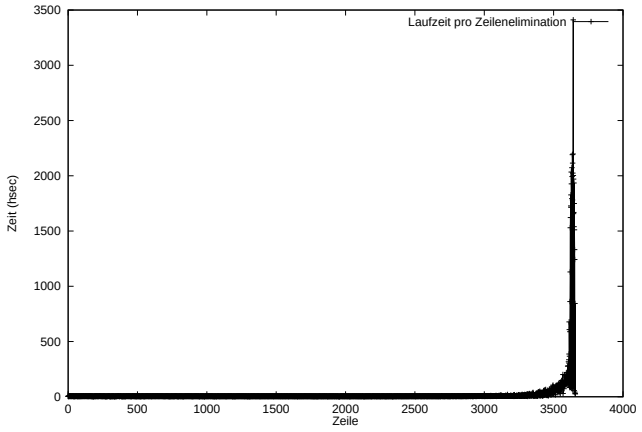


Abbildung 13.12: BSP_{Neis}: Laufzeit pro Iteration

Aufgrund dessen, daß beide der bisher zusammengestellten Hybrid–HNF–Algorithmus aktuell auf dem Datentyp *long* operieren, liegt das Abbruchkriterium für die aktuell betrachtete Berechnungsphase auf der Hand.

Abbruchkriterium: Übersteigt der maximale Eintrag einer Zwischenmatrix den maximal möglichen long–Eintrag der Maschine, so gehe zur nächsten Berechnungsphase über.

Nachdem das Abbruchkriterium festgelegt wurde, wird iterativ in die Betrachtung der nächsten Stufe übergegangen. Ist es schließlich möglich, eine HNF–Berechnung komplett durchzuführen, endet die Iteration und der Hybrid–HNF–Algorithmus wurde vollständig definiert.

13.3. Bemerkung

Für HNF–Hybrid–Algorithmen, die in späteren Stufen modulare HNF–Algorithmen einsetzen ist es oft sinnvoll, die Berechnung der Hadamard–Schranke in frühere Berechnungsphasen zu verlagern, um eine bessere, d.h. kleinere Schranke zu erhalten. Die Korrektheit der durchgeführten Berechnung ist davon nicht betroffen.

Kapitel 14

Ergebnisse

In diesem Kapitel stellen wir die gemäß der oben beschriebenen Vorgehensweise ermittelten Hybrid–HNF–Algorithmen für die Eingabematrixklassen $ClassGroup_{Jacobson}$ und $CLassGroup_{Neis}$ vor und illustrieren deren Effizienz anhand von Laufzeittabellen.

14.1 $ClassGroup_{Jacobson}$

Für Matrizen der Klasse $ClassGroup_{Jacobson}$ haben wir empirisch die folgende Ausführungsstrategie ermittelt. Im folgenden symbolisiert ML den maximal möglichen Long–Eintrag.

Strategie

1. [**Initialisierung**]
   In einem Initialisierungsschritt schaffen wir zunächst die Grundlagen für spätere Berechnungsphasen. Dazu berechnen wir die Hadamard–Schranke S der Relationenmatrix. Dies zu Beginn der HNF–Berechnung zu tun, ist sinnvoll, da mit fortschreitender HNF–Berechnung im allgemeinen die Größe der Einträge und damit die Hadamard–Schranke wächst.

2. [**HNF–Berechnung Phase 1**]
   Aufgrund der Beobachtung, daß die Einträge der Relationenmatrizen betragsmäßig durch 100 beschränkt sind, verwenden wir in der ersten Stufe der HNF–Berechnung eine Instantiierung auf der Basis des Datentyps *long*. Die geringe Eintragsdichte und die Verteilung der Einträge führt zur Auswahl des Algorithmus $(HNF_{Z2}, mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 1))$ ohne den finalen Matrixreduktionsschritt. Als Repräsentation verwenden wir die spaltenorientierte, dünnbesetzte Repräsentation.

   Abbruchkriterium: Die Ausführung dieser Strategie wird gestoppt, wenn der größte Eintrag die Schranke ML überschreitet.

3. **[HNF–Berechnung Phase 2]**
   Auf der zweiten Ebene verwenden wir eine Instantiierung auf das Basis des Datentyps *bigint*. Die Eintragsdichte und die Größe der Einträge ist im Vergleich zur Startmatrix deutlich angewachsen. Aus diesem Grund wird die Gewichtung in Richtung einer schnelleren Ausführung verschoben. Als Algorithmus wurde $(HNF_{Z1}, mgcd^{Pivot_1}_{BestRemainder})$ ausgewählt. Als Repräsentation verwenden wir in dieser Stufe die zeilenorientierte, dichtbesetzte Repräsentation.

   Abbruchkriterium: Die Ausführung dieser Strategie wird gestoppt, wenn der größte Eintrag die Schranke ML^4 überschreitet.

4. **[HNF–Berechnung Phase 3]**
   Zur Kontrolle der Zwischenergebnisse verwenden wir auf dieser Ebene für Matrizen mit vollem Zeilenrang eine modulare Vorgehensweise. Zunächst berechnen wir ein Vielfaches der Gitterdeterminanten von der zu verarbeitenden Restmatrix. Dazu verwenden wir die Methode $det_{CRT--Heuristik}$ auf der Basis des Algorithmus von Gauß–Bareiss. Als Parameter werden der Parameter $b = 5$ und die Schranke S verwendete, welche wir in der Initialisierungsphase berechnete haben. Anschließend führen wir den modularen HNF–Algorithmus $(HNFmod_{Z1}, mgcd_{linear})$ aus. Mit Hilfe dieses Algorithmus führen wir die HNF–Berechnung zu Ende. Besitzt die verbleibende Restmatrix nicht vollen Zeilenrang, greifen wir auf den nichtmodularen Algorithmus $(HNF_{KannanBachem}, Normalize_{ChouCollins})$ zurück.

5. **[Normalisierung]**
   Zur Reduktion der Nebendiagonalelemente verwenden wir aufgrund der besonderen Struktur der HNF den Normalisierungsalgorithmus $(NormalizeHybrid_{Std})$.

14.1. Bemerkung [LiDIA]
In der aktuellen Implementierung wird die Einhaltung der Schranken, ohne eine Überlauf im Long–Bereich in Kauf zu nehmen, mittels eines bigint–Arrays, in welchem der maximale Eintrag pro Spalte gehalten wird, realisiert.

Praktische Ergebnisse

Schauen wir uns zunächst die Laufzeit des obigen Hybrid–Verfahrens angewendet auf die ausgewählte Beispielmatrix der Eingabematrixklasse $ClassGroup_{Jacobson}$ an. Die Berech-

Matrix	Laufzeit		
	Phase 1	Phase 2	Gesamt
$BSP_{Jacobson}$	149 hsec	936 hsec	11297 hsec

Tabelle 14.1: Ergebnisse Hybrid–Vorgehensweise $ClassGroup_{Jacobson}$

nung konnte mit einer Laufzeit unter einer Stunde und mit einem Speicherplatzbedarf von weniger als 100 MB beendet werden. In den folgenden Tabellen haben wir weitere Ergebnisse zusammengestellt, die wir mit der obigen Strategie in Kombination mit der Algorithmus von Michael Jacobson Jr. ermittelt haben. Für weitere Details siehe [Jr.99].

Zeilen	Spalten	Laufzeit (CPU Sekunden)
80	100	0,09
90	113	0,11
100	230	0,34
110	153	0,18
120	145	0,29
130	175	0,41
140	185	0,45
150	196	0,57
160	208	0,73
180	206	1,20
200	223	2,05
220	247	1,71
240	296	4,49
260	286	2,42
325	378	6,07
355	413	12,14
375	438	12,70
400	481	13,37
425	519	28,91
550	622	21,12
650	741	37,54
750	829	75,00
850	925	97,39
950	1047	90,00
1000	1214	200,53
1150	1283	181,80
1300	1453	305,88
1600	1897	602,73
1900	2074	977,27
2200	2408	929,77
2500	2722	1670,80
2500	2917	1867,60
2500	3266	1697,75
3000	3209	4724,75
3000	3226	5459,17

Tabelle 14.2: Ergebniszusammenstellung I

Zeilen	Spalten	Laufzeit (CPU Sekunden)
80	103	0,08
90	112	0,11
100	147	0,16
110	132	0,14
120	165	0,24
130	175	0,31
140	164	0,28
150	175	0,38
160	184	0,73
180	207	1,10
200	227	0.82
220	264	1,49
240	288	1,64
260	309	2,16
325	379	5,04
355	414	7,35
375	491	9,99
400	458	10,21
425	474	18,42
550	611	24,01
650	754	39,19
750	986	70,53
850	956	66,40
950	1051	97,48
1000	1083	131,23
1150	1238	266,19
1300	1416	311,78
1600	1744	568,50
1900	2078	769,92
2200	2382	1108,01
2500	2725	1590,06
2500	2676	2737,50
2500	2676	1970,88
3000	4951	3671,59
3000	3198	5227,09

Tabelle 14.3: Ergebniszusammenstellung II

14.2 $ClassGroup_{Neis}$

Für Matrizen der Klasse $ClassGroup_{Neis}$ haben wir die folgende Strategie entwickelt, die sich stark an die für die Eingabematrixklasse $ClassGroup_{Jacobson}$ anlehnt. ML steht in der folgenden Beschreibung wiederum für den maximal möglichen Long–Eintrag.

1. [**Initialisierung**]
   In einem Initialisierungsschritt schaffen wir zunächst die Grundlagen für spätere Berechnungsphasen. Dazu berechnen wir die Hadamard–Schranke S der Relationenmatrix. Dies zu Beginn der HNF–Berechnung zu tun, ist sinnvoll, da mit fortschreitender HNF–Berechnung im Allgemeinen die Größe der Einträge und damit die Hadamard–Schranke wächst.

2. [**HNF–Berechnung Phase 1**]
   Aufgrund der Beobachtung, daß die Einträge der Relationenmatrizen betragsmäßig durch 100 beschränkt sind, verwenden wir in der ersten Stufe der HNF–Berechnung eine Instantiierung auf der Basis des Datentyps *long*. Die geringe Eintragsdichte und die Verteilung der Einträge führt zur Auswahl des Algorithmus $(HNF_{Z2}, mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 1))$ ohne den finalen Matrixreduktionsschritt. Als Repräsentation verwenden wir die spaltenorientierte, dünnbesetzte Repräsentation.

   Abbruchkriterium: Die Ausführung dieser Strategie wird gestoppt, wenn der größte Eintrag die Schranke ML überschreitet.

3. [**HNF–Berechnung Phase 2**]
   Auf der zweiten Ebene verwenden wir eine Instantiierung auf das Basis des Datentyps *bigint*. Die Eintragsdichte und die Größe der Einträge ist im Vergleich zur Startmatrix deutlich angewachsen. Aus diesem Grund wird die Gewichtung in Richtung einer schnelleren Ausführung verschoben. Als Algorithmus wurde $(HNF_{Z1}, mgcd^{Pivot_1}_{BestRemainder})$ ausgewählt. Als Repräsentation verwenden wir in dieser Stufe die zeilenorientierte, dichtbesetzte Repräsentation.

   Abbruchkriterium: Die Ausführung dieser Strategie wird gestoppt, wenn der größte Eintrag die Schranke ML^6 überschreitet.

4. [**HNF–Berechnung Phase 3**]
   Zur Kontrolle der Zwischenergebnisse verwenden wir auf dieser Ebene für Matrizen mit vollem Zeilenrang eine modulare Vorgehensweise. Zunächst berechnen wir ein Vielfaches der Gitterdeterminanten von der zu verarbeitenden Restmatrix. Dazu verwenden wir die Methode $det_{CRT--Heuristik}$ auf der Basis des Algorithmus von Gauß–Bareiss. Als Parameter werden der Parameter $b = 5$ und die Schranke S verwendete, welche wir in der Initialisierungsphase berechnete haben. Anschließend führen wir den modularen HNF–Algorithmus $(HNFmod_{Z1}, mgcd_{linear})$ aus. Mit Hilfe dieses Algorithmus führen wir die HNF–Berechnung zu Ende. Besitzt die verbleibende Restmatrix nicht vollen Zeilenrang, greifen wir auf den nichtmodularen Algorithmus $(HNF_{KannanBachem}, Normalize_{ChouCollins})$ zurück.

5. [**Normalisierung**]
   Zur Reduktion der Nebendiagonalelemente verwenden wir aufgrund der besonderen Struktur der HNF den Normalisierungsalgorithmus $(NormalizeHybrid_{ChouCollins})$.

Praktische Ergebnisse
Schauen wir uns zunächst die Ergebnisse dieses Algorithmus angewendet auf die obige Beispielmatrix der Matrixklasse $ClassGroup_{Neis}$ an. Innerhalb kürzester Zeit konnte die

Matrix	Laufzeit		
	Phase 1	Phase 2	Gesamt
BSP_{Neis}	64 hsec	–	220 hsec

Tabelle 14.4: Ergebnisse Hybrid–Vorgehensweise $ClassGroup_{Neis}$

HNF der Relationenmatrix errechnet werden. Auffällig ist, das die zweite Phase des HNF–Hybrid–Algorithmus keinen Beitrag zur Berechnung der HNF leisten mußte.

In den folgenden Tabellen haben wir eine Vielzahl weiterer Berechnungergebnisse aufgeführt, die zeigen, daß der entwickelte HNF–Hybridalgorithmus für diese Matrixklasse wirklich geeignet ist.

Zeilen	**Spalten**	**Laufzeit** (CPU Sekunden)
896	1931	107,20
807	1729	65,02
1129	2499	209,00
2961	6353	4083,64
2581	5566	1522,84
3393	7127	4995,03
3183	6821	4007,61
4455	9420	8558,01
4426	9420	7101,61
5551	11641	14104,58
5189	10827	12375,21
6643	13908	25416,05
6113	12789	16932,73
7197	14971	30037,91
7857	16182	40880,34
7802	16127	44105,32
7922	16434	37747,83
8352	17258	50942,81
8771	18242	5380278
8561	17844	50458,03
8538	17821	50793,35
9281	19352	66282,28
8941	18341	57551,16
9320	19323	64306,97
11913	24461	64674,35

Tabelle 14.5: Ergebniszusammenstellung III

Zeilen	**Spalten**	**Laufzeit** (CPU Sekunden)
204	436	101,00
139	307	0,60
583	1299	25,92
336	760	5,28
599	1293	26,68
435	941	11,43
677	1457	43,93
765	1681	42,58
743	1599	61,84
776	1632	80,24
827	1765	74,30
912	2003	106,63
893	1938	123,37
1058	2284	211,47
898	2011	115,52
1135	2445	212,34
1389	3051	554,62
1453	3115	529,38
1643	3562	521,65
2090	4551	113248
2297	4997	212306
2193	4767	1614,26
2377	5180	1935,73
2672	5689	2761,88
2905	6176	3123,92
3174	6952	4859,13
3080	6585	3730,34
3851	8010	5530,07
3868	8128	5341,41
3302	7067	3460,99
4114	8828	7725,58
4949	10249	8582,75
4813	10113	9190,21
4909	10360	10875,16
4823	10274	9220,62
6463	13555	22893,92
6216	12954	18968,28

Tabelle 14.6: Ergebniszusammenstellung IV

Zeilen	**Spalten**	**Laufzeit** (CPU Sekunden)
33	62	0,03
626	1406	77,19
123	291	0,67
534	1358	52,61
3678	8415	15998,99
4916	10509	16923,34
4088	9076	18733,37
5243	11989	39035,79
5147	11244	19937,21
7190	16330	101831,98
6290	14665	76845,80
6093	14003	68261,82
5417	12617	47795,93
8292	18735	152977,84
7132	16769	128422,32

Tabelle 14.7: Ergebniszusammenstellung V

Kapitel 15

Ausblick

Anhand der in dieser Arbeit dargestellten Überlegungen wird deutlich, daß es durchaus sinnvoll ist, aus bekannten Algorithmen Hybrid–Algorithmen für eine spezielle Aufgabenstellung herzuleiten. Es wird weiter deutlich, daß ein geeignetes Klassendesign, wie es in dieser Arbeit für die Matrixklassen in LiDIA und für die HNF–Algorithmen vorgestellt wurde, durchaus in der Lage ist, diesen Prozeß weitgehend zu unterstützen und zu vereinfachen. Leider ist die bisherige Vorgehensweise zur Konstruktion der Hybrid–Algorithmen noch aufwendig und an die natürliche Intelligenz, Intuition und Erfahrung des Anwenders gebunden.

Wünschenswert wären computergestützte, automatisierte Verfahren, die automatisch die zur Verfügung stehende Rechnerarchitektur und die Klasse der Eingabematrizen analysieren, die richtigen HNF–Bausteine und die richtigen Repräsentationen auswählen und diese so zusammensetzen können, so daß die HNF für die betrachteten Matrizen effizient auf der jeweiligen Hardwareplattform berechnet werden können. Dies wäre der logisch nächste Schritt zu Vervollständigung des vorgestellten Frameworks.

Teil VI

Anhang

Anhang A

Ergebniszusammenstellung: Mehrstufige Pivotkriterien

Dieser Abschnitt enthält die Laufzeitergebnisse der Modulalgorithmen zur Berechnung des *mgcd*, die sich mehrstufiger Pivotkriterien bedienen. Die Tabellen ergänzen die Betrachtungen des Abschnitts 10.1.1.

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	1	994.3	1	99	25.4873	1.039
450.1	90.02	99	1	944.6	1	99	23.6107	0.944
403.9	80.78	99	1	898.2	1	99	22.834	0.843
350.7	70.14	99	1	846.3	1	99	21.1815	0.74
295.6	59.12	98.9	1	791.8	1	98.9	18.9975	0.628
253.4	50.68	98.8	1.1	763.2	5.9	98.8	16.3878	0.576
198.7	39.74	98.7	1.4	736	19.4	97.7	12.5728	0.527
156.3	31.26	98.6	1.3	681.1	13.6	98.2	11.126	0.407
97	19.4	98.7	1.3	610	15.1	98.1	7.99262	0.263
51.6	10.32	98.1	1.9	579.6	29.4	95.4	4.40873	0.202
29.5	5.9	97.1	1.8	541.7	22.4	80.6	2.9856	0.126
11	2.2	93.6	2.5	521.2	11	45.3	1.52993	0.088
5.4	1.08	82.9	2.2	508.2	8.6	53.1	1.36403	0.065
2.8	0.56	62.7	2	504	8.4	34.2	1.13929	0.058
1.8	0.36	57.3	1.7	502	2.9	22.2	1.05976	0.051
0.5	0.1000000	18.8	1	500	1	1	1	0.045

Tabelle A.1: $mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	1.4	1114.1	169.6	995.7	205.27	1.328
449	89.8	997.6	1.5	1078.7	197.2	992.7	184.275	1.25
398.2	79.64	996	1.7	1093.9	206.5	986.8	155.614	1.277
348.9	69.78	996.6	1.7	1006.7	229.4	983.9	144.734	1.103
303.2	60.64	996.2	1.8	974.4	222.3	980.3	125.499	1.03
243.8	48.76	995.1	1.8	898.3	150.2	955.4	116.785	0.859
197.8	39.56	992.8	1.7	812.5	116.2	951.9	106.273	0.676
149	29.8	992.9	1.9	746.6	207.9	945.5	84.3077	0.538
98	19.6	987.6	1.9	667	154.8	874.2	55.6786	0.373
48.4	9.68	978.9	2	586.2	64.8	423	23.4186	0.215
25.7	5.14	962.9	2.6	556.3	119	418	10.3565	0.163
8.6	1.72	828.9	2.4	517.4	126.5	504.7	5.69946	0.081
6.2	1.24	887.6	3.1	514.8	121.9	455.7	4.06566	0.081
2.6	0.52	624.3	2.2	504.1	46.4	309.7	2.29677	0.063
2	0.4	567.1	1.9	503	69.2	235.2	1.95905	0.063
1.1	0.22	257.8	1.2	500.8	49.8	103.9	1.44688	0.052

Tabelle A.2: $mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2	1465	611	8780.1	1422.94	2.015
455.2	91.04	9985	2	1316.7	1937.8	9733	1439.7	1.738
397.3	79.46	9961.8	1.9	1234.9	426.8	8965.7	1368.74	1.541
352.3	70.46	9977.6	1.9	1135.6	864.4	7528.8	1094.33	1.34
301.7	60.34	9968.3	2	1072.1	910.5	8137.9	1063.28	1.211
252.5	50.5	9945.4	1.9	952.1	863.5	7674.5	945.195	0.96
197.8	39.56	9954.8	2.1	894.7	388.9	5457.3	583.655	0.84
151.5	30.3	9926	2	793.3	810.2	7764.6	698.577	0.642
99.9	19.98	9907.7	2.1	699.6	527.6	5786.1	385.732	0.451
51.9	10.38	9677.3	2.3	609.2	971.3	4891.9	193.146	0.27
26.4	5.28	9739.2	3	571.6	260.7	1896.3	37.096	0.193
12.8	2.56	9122.1	3.2	535.1	511.7	2548.9	30.9873	0.115
4.2	0.84	8115.2	2.9	509.5	678.8	2817.9	13.9211	0.076
3.9	0.78	8317.4	2.9	508.4	396.1	2393.6	13.1635	0.075
1.7	0.34	4758.2	1.7	503	371.4	1391.7	8.40179	0.057
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.053

Tabelle A.3: $mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.1	1509.2	2111.7	65063.3	10409.7	2.11
452.1	90.42	99823.7	2	1396.5	918.1	68139.4	10987	1.866
399.3	79.86	99763.5	2.2	1336	3136.7	61039	8525.07	1.757
352.9	70.58	99657.1	2.1	1223.9	3370.8	61260.6	8774.67	1.571
307.4	61.48	99466.2	2.1	1137.5	2072.4	61931.5	8049.9	1.385
251.8	50.36	99431.8	2.1	1020.4	7155.2	49672.6	5985.42	1.16
203.4	40.68	99511.4	2.8	1031.7	6300.6	34603.1	3027.43	1.146
150.7	30.14	99297.6	2.6	877.4	2409.8	31808.8	2416.62	0.8
104.3	20.86	98793.9	2.6	754.6	5390.8	30449.6	1832.88	0.579
54.8	10.96	97747.7	3.3	666.3	4849.2	26893	927.192	0.37
24.6	4.92	96337.4	3.5	577.1	1990.4	17775.4	324.096	0.202
11.5	2.3	92767.5	3.6	536.4	2280.2	18471.7	191.83	0.127
5.1	1.02	79355.4	3.7	515.4	2194	6676.2	42.6019	0.092
2.3	0.46	58436	2.1	504.5	3126.9	10555.9	46.3663	0.072
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.052
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.051

Tabelle A.4: $mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	1	994.3	1	99	25.4873	1.045
450.1	90.02	99	1	944.6	1	99	23.6107	0.947
403.9	80.78	99	1	898.2	1	99	22.834	0.848
350.7	70.14	99	1	846.3	1	99	21.1815	0.742
295.6	59.12	98.9	1	791.8	1	98.9	18.9975	0.629
253.4	50.68	98.8	1.1	763.2	1	94	15.9553	0.59
198.7	39.74	98.7	1.4	735.9	1.2	79.5	11.2519	0.574
156.3	31.26	98.6	1.3	681.1	1.1	82.8	9.98209	0.429
97	19.4	98.7	1.3	610.3	1.3	84.2	7.44159	0.273
51.6	10.32	98.1	1.9	579.6	2.1	42.8	2.81159	0.212
29.5	5.9	97.1	1.8	542	4.1	53.9	2.43303	0.131
11	2.2	93.6	2.5	521.4	6.5	38.7	1.4697	0.09
5.4	1.08	82.9	2.2	508.2	8.6	53.1	1.36403	0.066
2.8	0.56	62.7	1.9	503.8	7.8	33.1	1.13438	0.059
1.8	0.36	57.3	1.7	502	2.9	22.2	1.05976	0.054
0.5	0.1000000	18.8	1	500	1	1	1	0.045

Tabelle A.5: $mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	1.4	1114.1	1.5	769.1	173.992	1.465
449	89.8	997.6	1.5	1078.6	2.1	699.1	147.847	1.399
398.2	79.64	996	1.7	1093.7	3.6	514.9	92.9439	1.416
348.9	69.78	996.6	1.7	1006.3	3.1	537.1	92.8397	1.231
303.2	60.64	996.2	1.8	974	3.5	430	64.8259	1.128
243.8	48.76	995.1	1.8	897.9	5.4	379.5	51.1132	0.91
197.8	39.56	992.8	1.7	812.4	3	442	54.6108	0.704
149	29.8	992.9	1.9	746.3	7.1	346.4	36.6028	0.576
98	19.6	987.6	1.9	667	7.5	360	26.4529	0.387
48.4	9.68	978.9	2	586	7.3	434.3	19.1355	0.219
25.7	5.14	962.9	2.8	558.8	13.7	250.9	7.42269	0.167
8.6	1.72	828.9	2.5	518	84.2	445.7	4.84479	0.083
6.2	1.24	887.6	3.1	515.6	54.1	311.6	3.26299	0.083
2.6	0.52	624.3	2.2	504.1	46.4	309.7	2.29677	0.06
2	0.4	567.1	2	503.3	58.8	207.1	1.8025	0.058
1.1	0.22	257.8	1.3	501	46	96.3	1.41517	0.053

Tabelle A.6: $mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2	1464.5	17.1	1004.1	144.562	2.056
455.2	91.04	9985	2	1316.4	26.3	2129.8	349.76	1.882
397.3	79.46	9961.8	1.9	1234.8	21.5	1988.5	288.863	1.575
352.3	70.46	9977.6	1.9	1135.2	10.6	2338.1	346.663	1.389
301.7	60.34	9968.3	2	1071.8	20.5	1428.5	183.141	1.256
252.5	50.5	9945.4	1.9	951.8	26.5	2717.9	367.192	0.994
197.8	39.56	9954.8	2	894.2	18.8	1852	211.052	0.867
151.5	30.3	9926	2.1	799.3	50.8	2221.8	194.117	0.661
99.9	19.98	9907.7	2.2	707	56.7	3406.7	227.678	0.463
51.9	10.38	9677.3	2.5	614.4	94.5	3119.6	126.759	0.277
26.4	5.28	9739.2	3.2	575	102.4	1336.1	27.6783	0.199
12.8	2.56	9122.1	3.2	534.8	261.5	2027.2	27.5075	0.12
4.2	0.84	8115.2	3	509.8	609.8	2171.8	10.9249	0.078
3.9	0.78	8317.4	2.9	508.4	396.1	2393.6	13.1635	0.077
1.7	0.34	4758.2	1.8	503.5	322.6	1073.2	5.97994	0.055
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.054

Tabelle A.7: $mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.1	1509	185.6	13911.2	2105.63	2.164
452.1	90.42	99823.7	2	1396.5	136.7	15637.2	2505.28	1.872
399.3	79.86	99763.5	2.2	1336.1	180.4	23974.4	3469.47	1.811
352.9	70.58	99657.1	2.2	1241.7	296.7	43562.8	6096.67	1.626
307.4	61.48	99466.2	2.2	1158.7	278.3	29806.6	3855.83	1.412
251.8	50.36	99431.8	2.3	1045.8	348.8	31297.1	3636.1	1.208
203.4	40.68	99511.4	2.9	1040.5	403.7	15011.6	1284.9	1.221
150.7	30.14	99297.6	2.6	877.2	216.5	15794.7	1294.67	0.816
104.3	20.86	98793.9	2.8	770	436.3	12465.8	770.605	0.597
54.8	10.96	97747.7	3.3	665.8	658.5	13612	530.167	0.379
24.6	4.92	96337.4	3.7	580.2	486.2	8308.4	167.912	0.203
11.5	2.3	92767.5	3.6	536.4	1690.6	15939.8	165.635	0.127
5.1	1.02	79355.4	3.8	515.6	1005.2	4878.9	35.9884	0.096
2.3	0.46	58436	2.2	504.7	1908.9	9678.4	37.1631	0.071
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.047
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.049

Tabelle A.8: $mgcd^{Pivot_{1+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	1	994.3	1	99	25.4873	1.041
450.1	90.02	99	1	944.6	1	99	23.6107	0.939
403.9	80.78	99	1	898.2	1	99	22.834	0.846
350.7	70.14	99	1	846.3	1	99	21.1815	0.733
295.6	59.12	98.9	1	791.8	1	98.9	18.9975	0.627
253.4	50.68	98.8	1.1	763.2	2.3	95.3	16.0142	0.573
198.7	39.74	98.7	1.4	735.9	14.4	92.6	12.1307	0.531
156.3	31.26	98.6	1.3	681.5	11	93.4	10.5859	0.408
97	19.4	98.7	1.3	610.1	7.1	90	7.62842	0.263
51.6	10.32	98.1	1.9	579.4	15.8	63.5	3.41422	0.201
29.5	5.9	97.1	1.8	541.9	15.8	73.1	2.85219	0.126
11	2.2	93.6	2.3	520.2	11.9	53.8	1.61688	0.09
5.4	1.08	82.9	2.2	508.2	10.4	58.5	1.39984	0.067
2.8	0.56	62.7	2	503.8	7.9	35.2	1.1441	0.058
1.8	0.36	57.3	1.7	502	2.9	22.2	1.05976	0.052
0.5	0.1000000	18.8	1	500	1	1	1	0.047

Tabelle A.9: $mgcd^{Pivot_{1+4}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	1.4	1114	49.2	827.5	177.669	1.347
449	89.8	997.6	1.5	1078.5	125.3	868	161.88	1.272
398.2	79.64	996	1.7	1093.9	131.7	777.5	120.265	1.3
348.9	69.78	996.6	1.7	1006.3	106.3	742.6	111.122	1.123
303.2	60.64	996.2	1.8	974.5	96.7	691.3	89.8586	1.043
243.8	48.76	995.1	1.8	898.2	100.7	737.8	88.2621	0.865
197.8	39.56	992.8	1.7	812.4	62.6	697.7	76.7229	0.678
149	29.8	992.9	1.9	746.4	143.6	683.5	60.0075	0.543
98	19.6	987.6	1.9	667	128.3	739.3	46.7982	0.374
48.4	9.68	978.9	1.8	581	72.9	570.2	22.7363	0.214
25.7	5.14	962.9	2.4	554.2	133.5	451.8	11.3941	0.164
8.6	1.72	828.9	2.3	517.1	147.2	523.6	5.82692	0.082
6.2	1.24	887.6	2.8	514.4	124.7	464.2	4.12811	0.082
2.6	0.52	624.3	2.2	504.1	46.4	309.7	2.29677	0.061
2	0.4	567.1	1.9	503	69.2	235.2	1.95905	0.06
1.1	0.22	257.8	1.2	500.8	49.8	103.9	1.44688	0.05

Tabelle A.10: $mgcd^{Pivot_{1+4}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	2	1465	322.1	5609.6	890.969	2.023
455.2	91.04	9985	2	1316.6	1388.8	7213.6	1031.97	1.757
397.3	79.46	9961.8	1.9	1234.8	182.8	5406.4	820.738	1.544
352.3	70.46	9977.6	1.9	1135.5	292.7	7151.6	1082.79	1.349
301.7	60.34	9968.3	2	1072.1	334.6	5861.3	771.737	1.218
252.5	50.5	9945.4	1.9	952.1	610.1	7649.2	964.346	0.963
197.8	39.56	9954.8	2	884.5	231.6	4272.4	458.584	0.842
151.5	30.3	9926	2	793.3	597.5	6462.6	580.517	0.642
99.9	19.98	9907.7	2.1	699.7	364.8	5631	385.484	0.453
51.9	10.38	9677.3	2.1	602.8	1306.5	5921.9	225.34	0.27
26.4	5.28	9739.2	3	572.1	484.2	2576.9	45.5466	0.192
12.8	2.56	9122.1	3.1	535	498.4	2760	36.3379	0.114
4.2	0.84	8115.2	2.9	509.5	632.8	2757.7	13.5812	0.074
3.9	0.78	8317.4	2.9	508.4	445.8	2526.1	13.8845	0.075
1.7	0.34	4758.2	1.6	502.6	788	1924	11.0217	0.059
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.053

Tabelle A.11: $mgcd^{Pivot_{1+4}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	2.1	1508.9	1460.3	46915.7	7399.19	2.116
452.1	90.42	99823.7	2	1396.5	381.6	36463.9	5889.05	1.865
399.3	79.86	99763.5	2.1	1319.2	5281.6	44099.2	5696.54	1.76
352.9	70.58	99657.1	2.1	1224.3	4161.1	57489.5	8102.03	1.578
307.4	61.48	99466.2	2.1	1137.7	2266.8	60104.4	7563.53	1.386
251.8	50.36	99431.8	2.1	1021.3	7557.2	48556.6	5807.93	1.168
203.4	40.68	99511.4	2.7	1017	8123	31869.9	2464.25	1.155
150.7	30.14	99297.6	2.6	878.1	927.3	23451.3	1872.25	0.799
104.3	20.86	98793.9	2.6	754.5	3745.8	23871.2	1475.11	0.58
54.8	10.96	97747.7	3.1	659.5	1447.1	15520.6	592.062	0.369
24.6	4.92	96337.4	3.4	577.1	1655.2	16910.9	334.31	0.201
11.5	2.3	92767.5	3.4	534.7	8479.8	27635.4	242.517	0.125
5.1	1.02	79355.4	3.7	515.2	2183.3	6650	42.4247	0.09
2.3	0.46	58436	2.1	504.5	3319.8	11134.6	49.9374	0.067
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.05
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.051

Tabelle A.12: $mgcd^{Pivot_{1+4}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2	1798.9	5.3	37.1	6.50342	3.358
450.1	90.02	99	1.9	1665.2	5	35.1	6.01742	3.051
403.9	80.78	99	2	1520	8	37.8	7.15507	2.707
350.7	70.14	99	1.9	1403	3.1	38.2	6.3784	2.364
295.6	59.12	98.9	2	1215.8	16.7	46	6.74963	1.836
253.4	50.68	98.8	2.1	1077.3	10.7	52.1	8.85102	1.569
198.7	39.74	98.7	2.1	992.3	4.8	46.3	6.66522	1.254
156.3	31.26	98.6	2.1	921.6	6.8	34.2	4.77094	1.096
97	19.4	98.7	2	736	6.7	36.6	3.55978	0.636
51.6	10.32	98.1	2.5	627.9	3.6	35.3	3.22663	0.329
29.5	5.9	97.1	2.7	578.8	7.4	33.3	2.1764	0.22
11	2.2	93.6	3.1	528.1	4.3	28	1.46582	0.109
5.4	1.08	82.9	2.6	510.3	14.6	52	1.44876	0.072
2.8	0.56	62.7	2	504.6	8.5	29.7	1.13099	0.061
1.8	0.36	57.3	1.8	502.1	6	21.1	1.06811	0.053
0.5	0.1000000	18.8	1	500	1	1	1	0.045

Tabelle A.13: $mgcd^{Pivot_{2+1}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.4	2209.9	26	247.7	34.4333	4.504
449	89.8	997.6	2.5	2254.1	5.6	62.3	6.70769	4.614
398.2	79.64	996	2.4	1981.2	2.5	56.4	9.45134	3.966
348.9	69.78	996.6	2.4	1648.4	13.9	169.7	29.2583	3.202
303.2	60.64	996.2	2.9	1635.9	4.8	90	14.5302	3.063
243.8	48.76	995.1	3.2	1421.1	14.3	116.4	17.5916	2.386
197.8	39.56	992.8	2.9	1244.9	15.8	152.8	20.1573	1.976
149	29.8	992.9	2.5	969.4	5.8	147.2	18.2803	1.223
98	19.6	987.6	2.8	825.2	9	192.7	15.2665	0.866
48.4	9.68	978.9	2.8	668	11.1	151.5	9.0774	0.422
25.7	5.14	962.9	3.4	576.5	25	176.2	6.95733	0.224
8.6	1.72	828.9	3.5	526.4	46.9	250.6	2.99145	0.103
6.2	1.24	887.6	3.6	518	72.5	219.6	2.7888	0.095
2.6	0.52	624.3	2.4	504.3	45	290.9	2.24093	0.065
2	0.4	567.1	1.9	503	88.2	282.3	2.2497	0.06
1.1	0.22	257.8	1.4	501.2	12.4	25.7	1.19453	0.048

Tabelle A.14: $mgcd^{Pivot_{2+1}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	3.3	2837.7	4.7	488.3	64.358	6.177
455.2	91.04	9985	3.3	2552.5	5.4	275.6	30.8352	5.531
397.3	79.46	9961.8	3.5	2116.7	191.3	955.5	98.4669	4.294
352.3	70.46	9977.6	3.6	2073.7	6.2	227.2	30.1844	4.16
301.7	60.34	9968.3	3.3	1844.5	3.1	567.7	73.1039	3.534
252.5	50.5	9945.4	4.2	1699.1	47.4	372.1	35.2389	3.181
197.8	39.56	9954.8	4.9	1395.3	9.9	240.1	25.6489	2.361
151.5	30.3	9926	4.2	1213.1	13.6	205.6	20.6342	1.899
99.9	19.98	9907.7	4.6	1001	8.6	212	14.8139	1.214
51.9	10.38	9677.3	4.4	717.4	37.4	318.7	16.6543	0.577
26.4	5.28	9739.2	4.6	620.3	21	257	9.19233	0.32
12.8	2.56	9122.1	3.9	551.7	39.8	514.3	6.49375	0.177
4.2	0.84	8115.2	3.6	512.6	479.6	1758.4	10.1797	0.087
3.9	0.78	8317.4	3	509.6	321.8	1366	7.34125	0.085
1.7	0.34	4758.2	1.7	503	371.4	1554.7	9.75388	0.059
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.051

Tabelle A.15: $mgcd^{Pivot_{2+1}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	4.5	3232	12.4	835.9	86.3924	7.216
452.1	90.42	99823.7	4	2903.7	25.8	490.4	49.1691	6.376
399.3	79.86	99763.5	5.5	3042.2	5.3	239.9	19.6308	6.695
352.9	70.58	99657.1	5.5	2701.5	8	682.2	38.4588	5.324
307.4	61.48	99466.2	5	2194.3	99.7	980.3	80.876	4.622
251.8	50.36	99431.8	4.8	2015.2	17.5	289	25.6866	3.776
203.4	40.68	99511.4	5.2	1631.8	12.4	569.1	40.4026	3.025
150.7	30.14	99297.6	5.4	1340.6	16.6	6456.7	583.818	2.219
104.3	20.86	98793.9	5.5	1101.6	32.5	9952.4	1012.96	1.5
54.8	10.96	97747.7	6.3	838.8	19.7	339.2	22.6059	0.862
24.6	4.92	96337.4	5	615.9	63.4	1026.4	23.6365	0.347
11.5	2.3	92767.5	5.1	553.3	456.2	5654.2	77.2304	0.188
5.1	1.02	79355.4	4	517.4	1022.5	4040.7	22.7574	0.109
2.3	0.46	58436	2.3	504.9	1710	9141.5	31.3012	0.075
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.048
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.05

Tabelle A.16: $mgcd^{Pivot_{2+1}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2	1798.9	5.3	37.1	6.50342	3.362
450.1	90.02	99	1.9	1665.2	5	35.1	6.01742	3.058
403.9	80.78	99	2	1520	8	37.8	7.15507	2.718
350.7	70.14	99	1.9	1403	3.1	38.2	6.3784	2.371
295.6	59.12	98.9	2	1215.8	16.7	46	6.74963	1.848
253.4	50.68	98.8	2.1	1077.3	10.7	52.1	8.85102	1.571
198.7	39.74	98.7	2.1	992.3	4.8	46.3	6.66522	1.258
156.3	31.26	98.6	2.1	921.6	6.8	34.2	4.77094	1.101
97	19.4	98.7	2	736	6.7	36.6	3.55978	0.636
51.6	10.32	98.1	2.5	627.9	3.6	35.3	3.22663	0.333
29.5	5.9	97.1	2.7	578.8	7.4	33.3	2.1764	0.221
11	2.2	93.6	3.1	528.1	4.3	28	1.46582	0.116
5.4	1.08	82.9	2.6	510.3	14.6	52	1.44876	0.073
2.8	0.56	62.7	2	504.6	8.5	29.7	1.13099	0.06
1.8	0.36	57.3	1.8	502.1	6	21.1	1.06811	0.053
0.5	0.1000000	18.8	1	500	1	1	1	0.047

Tabelle A.17: $mgcd^{Pivot_{2+3}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.4	2209.9	26	247.7	34.4333	4.506
449	89.8	997.6	2.5	2254.1	5.6	62.3	6.70769	4.616
398.2	79.64	996	2.4	1981.2	2.5	56.4	9.45134	3.972
348.9	69.78	996.6	2.4	1648.4	13.9	169.7	29.2583	3.207
303.2	60.64	996.2	2.9	1635.9	4.8	90	14.5302	3.071
243.8	48.76	995.1	3.2	1421.1	14.3	116.4	17.5916	2.387
197.8	39.56	992.8	2.9	1244.9	15.8	152.8	20.1573	1.976
149	29.8	992.9	2.5	969.4	5.8	147.2	18.2803	1.226
98	19.6	987.6	2.8	825.2	9	192.7	15.2665	0.867
48.4	9.68	978.9	2.8	668	11.1	151.5	9.0774	0.424
25.7	5.14	962.9	3.4	576.5	25	176.2	6.95733	0.224
8.6	1.72	828.9	3.5	526.4	46.9	250.6	2.99145	0.106
6.2	1.24	887.6	3.6	518	72.5	219.6	2.7888	0.095
2.6	0.52	624.3	2.4	504.3	45	290.9	2.24093	0.066
2	0.4	567.1	1.9	503	88.2	282.3	2.2497	0.058
1.1	0.22	257.8	1.4	501.2	12.4	25.7	1.19453	0.053

Tabelle A.18: $mgcd^{Pivot_{2+3}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	3.3	2837.7	4.7	488.3	64.358	6.184
455.2	91.04	9985	3.3	2552.5	5.4	275.6	30.8352	5.543
397.3	79.46	9961.8	3.5	2116.7	191.3	955.5	98.4669	4.305
352.3	70.46	9977.6	3.6	2073.7	6.2	227.2	30.1844	4.161
301.7	60.34	9968.3	3.3	1844.5	3.1	567.7	73.1039	3.54
252.5	50.5	9945.4	4.2	1699.1	47.4	372.1	35.2389	3.192
197.8	39.56	9954.8	4.9	1395.3	9.9	240.1	25.6489	2.369
151.5	30.3	9926	4.2	1213.1	13.6	205.6	20.6342	1.904
99.9	19.98	9907.7	4.6	1001	8.6	212	14.8139	1.22
51.9	10.38	9677.3	4.4	717.4	37.4	318.7	16.6543	0.583
26.4	5.28	9739.2	4.6	620.3	21	257	9.19233	0.32
12.8	2.56	9122.1	3.9	551.7	39.8	514.3	6.49375	0.182
4.2	0.84	8115.2	3.6	512.6	479.6	1758.4	10.1797	0.088
3.9	0.78	8317.4	3	509.6	321.8	1366	7.34125	0.084
1.7	0.34	4758.2	1.7	503	371.4	1554.7	9.75388	0.058
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.052

Tabelle A.19: $mgcd^{Pivot_{2+3}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	4.5	3232	12.4	835.9	86.3924	7.226
452.1	90.42	99823.7	4	2903.7	25.8	490.4	49.1691	6.384
399.3	79.86	99763.5	5.5	3042.2	5.3	239.9	19.6308	6.706
352.9	70.58	99657.1	5.5	2701.5	8	682.2	38.4588	5.313
307.4	61.48	99466.2	5	2194.3	99.7	980.3	80.876	4.63
251.8	50.36	99431.8	4.8	2015.2	17.5	289	25.6866	3.784
203.4	40.68	99511.4	5.2	1631.8	12.4	569.1	40.4026	3.032
150.7	30.14	99297.6	5.4	1340.6	16.6	6456.7	583.818	2.223
104.3	20.86	98793.9	5.5	1101.6	32.5	9952.4	1012.96	1.507
54.8	10.96	97747.7	6.3	838.8	19.7	339.2	22.6059	0.865
24.6	4.92	96337.4	5	615.9	63.4	1026.4	23.6365	0.35
11.5	2.3	92767.5	5.1	553.3	456.2	5654.2	77.2304	0.192
5.1	1.02	79355.4	4	517.4	1022.5	4040.7	22.7574	0.112
2.3	0.46	58436	2.3	504.9	1710	9141.5	31.3012	0.076
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.053
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.053

Tabelle A.20: $mgcd^{Pivot_{2+3}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2	1798.9	5.3	37.1	6.50342	4.682
450.1	90.02	99	1.9	1665.2	5	35.1	6.01742	4.159
403.9	80.78	99	2	1520	8	37.8	7.15507	3.68
350.7	70.14	99	1.9	1403	3.1	38.2	6.3784	3.056
295.6	59.12	98.9	2	1215.8	16.7	46	6.74963	2.446
253.4	50.68	98.8	2.1	1077.3	10.7	52.1	8.85102	2.009
198.7	39.74	98.7	2.1	992.3	4.8	46.3	6.66522	1.577
156.3	31.26	98.6	2.1	921.6	6.8	34.2	4.77094	1.375
97	19.4	98.7	2	736	6.7	36.6	3.55978	0.766
51.6	10.32	98.1	2.5	627.9	3.6	35.3	3.22663	0.371
29.5	5.9	97.1	2.7	578.8	7.4	33.3	2.1764	0.24
11	2.2	93.6	3.1	528.1	4.3	28	1.46582	0.118
5.4	1.08	82.9	2.6	510.3	14.6	52	1.44876	0.076
2.8	0.56	62.7	2	504.6	8.5	29.7	1.13099	0.063
1.8	0.36	57.3	1.8	502.1	6	21.1	1.06811	0.057
0.5	0.1000000	18.8	1	500	1	1	1	0.048

Tabelle A.21: $mgcd^{Pivot_{2+4}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.4	2209.9	26	247.7	34.4333	6.231
449	89.8	997.6	2.5	2254.1	5.6	62.3	6.70769	6.674
398.2	79.64	996	2.4	1981.2	2.5	56.4	9.45134	5.584
348.9	69.78	996.6	2.4	1648.4	13.9	169.7	29.2583	4.311
303.2	60.64	996.2	2.9	1635.9	4.8	90	14.5302	4.095
243.8	48.76	995.1	3.2	1421.1	14.3	116.4	17.5916	3.171
197.8	39.56	992.8	2.9	1244.9	15.8	152.8	20.1573	2.601
149	29.8	992.9	2.5	969.4	5.8	147.2	18.2803	1.485
98	19.6	987.6	2.8	825.2	9	192.7	15.2665	1.051
48.4	9.68	978.9	2.8	668	11.1	151.5	9.0774	0.481
25.7	5.14	962.9	3.4	576.5	25	176.2	6.95733	0.244
8.6	1.72	828.9	3.5	526.4	46.9	250.6	2.99145	0.109
6.2	1.24	887.6	3.6	518	72.5	219.6	2.7888	0.102
2.6	0.52	624.3	2.4	504.3	45	290.9	2.24093	0.0671
2	0.4	567.1	1.9	503	88.2	282.3	2.2497	0.0631
1.1	0.22	257.8	1.4	501.2	12.4	25.7	1.19453	0.0521

Tabelle A.22: $mgcd^{Pivot_{2+4}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	3.3	2837.7	4.7	488.3	64.358	8.845
455.2	91.04	9985	3.3	2552.5	5.4	275.6	30.8352	7.854
397.3	79.46	9961.8	3.5	2116.7	191.3	955.5	98.4669	5.781
352.3	70.46	9977.6	3.6	2073.7	6.2	227.2	30.1844	5.489
301.7	60.34	9968.3	3.3	1844.5	3.1	567.7	73.1039	4.791
252.5	50.5	9945.4	4.2	1699.1	47.4	372.1	35.2389	4.321
197.8	39.56	9954.8	4.9	1395.3	9.9	240.1	25.6489	3.083
151.5	30.3	9926	4.2	1213.1	13.6	205.6	20.6342	2.447
99.9	19.98	9907.7	4.6	1001	8.6	212	14.8139	1.509
51.9	10.38	9677.3	4.4	717.4	37.4	318.7	16.6543	0.664
26.4	5.28	9739.2	4.6	620.3	21	257	9.19233	0.361
12.8	2.56	9122.1	3.9	551.7	39.8	514.3	6.49375	0.199
4.2	0.84	8115.2	3.6	512.6	479.6	1758.4	10.1797	0.093
3.9	0.78	8317.4	3	509.6	321.8	1366	7.34125	0.088
1.7	0.34	4758.2	1.7	503	371.4	1554.7	9.75388	0.062
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.052

Tabelle A.23: $mgcd^{Pivot_{2+4}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	4.5	3232	12.4	835.9	86.3924	9.862
452.1	90.42	99823.7	4	2903.7	25.8	490.4	49.1691	8.722
399.3	79.86	99763.5	5.5	3042.2	5.3	239.9	19.6308	9.709
352.9	70.58	99657.1	5.5	2701.5	8	682.2	38.4588	7.259
307.4	61.48	99466.2	5	2194.3	99.7	980.3	80.876	6.308
251.8	50.36	99431.8	4.8	2015.2	17.5	289	25.6866	5.121
203.4	40.68	99511.4	5.2	1631.8	12.4	569.1	40.4026	4.023
150.7	30.14	99297.6	5.4	1340.6	16.6	6456.7	583.818	2.85
104.3	20.86	98793.9	5.5	1101.6	32.5	9952.4	1012.96	1.906
54.8	10.96	97747.7	6.3	838.8	19.7	339.2	22.6059	1.032
24.6	4.92	96337.4	5	615.9	63.4	1026.4	23.6365	0.387
11.5	2.3	92767.5	5.1	553.3	456.2	5654.2	77.2304	0.209
5.1	1.02	79355.4	4	517.4	1022.5	4040.7	22.7574	0.116
2.3	0.46	58436	2.3	504.9	1710	9141.5	31.3012	0.08
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.051
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.051

Tabelle A.24: $mgcd^{Pivot_{2+4}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2.6	1452.9	1	2	1.0139	2.224
450.1	90.02	99	2.6	1372.6	1	2.3	1.01552	2.049
403.9	80.78	99	2.3	1294.1	1	2	1.01035	1.875
350.7	70.14	99	2.7	1225	1	2.2	1.02041	1.668
295.6	59.12	98.9	2.6	1135.8	1	2.4	1.02051	1.471
253.4	50.68	98.8	2.1	1056.4	1	2.1	1.0088	1.29
198.7	39.74	98.7	2.5	972.1	1	2.2	1.01543	1.077
156.3	31.26	98.6	2.6	905.4	1	2.4	1.01833	0.871
97	19.4	98.7	3.1	785.4	1	2.6	1.01999	0.602
51.6	10.32	98.1	3.3	675.1	1	2.7	1.02266	0.364
29.5	5.9	97.1	3.4	605.4	1.1	2.5	1.02494	0.242
11	2.2	93.6	5.2	547.4	2.3	7.9	1.08221	0.137
5.4	1.08	82.9	3.7	516.8	6.3	20.8	1.13777	0.091
2.8	0.56	62.7	2.3	505.4	9.1	30.2	1.11694	0.064
1.8	0.36	57.3	1.7	502.2	2.6	13.4	1.0462	0.077
0.5	0.1000000	18.8	1	500	1	1	1	0.088

Tabelle A.25: $mgcd^{Pivot_{3+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.8	2172	1	2.3	1.01611	3.743
449	89.8	997.6	4	2128.8	1.1	2.9	1.03514	3.357
398.2	79.64	996	3.9	1959.1	1.1	2.9	1.03787	3.03
348.9	69.78	996.6	4.4	1813.4	1	2.6	1.02873	2.678
303.2	60.64	996.2	4.1	1633.6	1.1	3.1	1.03838	2.354
243.8	48.76	995.1	3.7	1429.1	1.1	3	1.03408	1.894
197.8	39.56	992.8	3.6	1261.5	1.1	2.6	1.02822	1.573
149	29.8	992.9	4.5	1131.8	1.3	3.2	1.03949	1.201
98	19.6	987.6	5	971.1	1.4	3.7	1.0485	0.829
48.4	9.68	978.9	5.2	779.1	1.6	4.1	1.06084	0.479
25.7	5.14	962.9	7.5	685.8	1.7	4.6	1.07437	0.302
8.6	1.72	828.9	6.1	546.7	2.9	8	1.11304	0.149
6.2	1.24	887.6	5.1	529.7	34.4	85.7	1.47631	0.12
2.6	0.52	624.3	2.6	505.3	46.4	244.6	1.91094	0.081
2	0.4	567.1	1.9	503.1	71.5	222.1	1.85669	0.076
1.1	0.22	257.8	1.4	501.4	5	13.9	1.05943	0.077

Tabelle A.26: $mgcd^{Pivot_{3+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	4.3	2591.8	1.1	2.9	1.03376	4.511
455.2	91.04	9985	4.8	2477.8	1	2.8	1.03273	4.126
397.3	79.46	9961.8	4.7	2315.7	1.1	3	1.03766	3.647
352.3	70.46	9977.6	4.9	2131.5	1	3	1.03167	3.288
301.7	60.34	9968.3	4.8	1970.5	1.1	3	1.03816	2.861
252.5	50.5	9945.4	4.9	1813.6	1	3	1.03667	2.453
197.8	39.56	9954.8	4.2	1570.2	1	3	1.0321	1.986
151.5	30.3	9926	5.3	1457.5	1.2	3.7	1.0483	1.575
99.9	19.98	9907.7	6.3	1243.6	1.3	3.9	1.05002	1.112
51.9	10.38	9677.3	8.8	974.1	1.7	5	1.07648	0.647
26.4	5.28	9739.2	9.8	775.2	2.4	6.1	1.12758	0.394
12.8	2.56	9122.1	9.4	612.5	3.5	13.2	1.26531	0.23
4.2	0.84	8115.2	4.1	515.5	405.3	964.3	4.7579	0.116
3.9	0.78	8317.4	3.7	512.6	384.9	1441.8	9.93894	0.106
1.7	0.34	4758.2	2	504.1	315.1	954.7	5.21762	0.098
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.071

Tabelle A.27: $mgcd^{Pivot_{3+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	5.3	3660.6	1.2	3.8	1.04718	5.67
452.1	90.42	99823.7	6.9	3716.9	1.5	4.2	1.05462	5.214
399.3	79.86	99763.5	5.3	3193.9	1.2	3.9	1.04737	4.689
352.9	70.58	99657.1	5.8	3046.3	1.1	4	1.04816	4.249
307.4	61.48	99466.2	6.7	2830.9	1.3	4	1.04013	3.722
251.8	50.36	99431.8	7.3	2547.6	1.2	4	1.04483	3.112
203.4	40.68	99511.4	8.6	2289.7	1.5	4.6	1.05813	2.546
150.7	30.14	99297.6	9	1968.6	1.4	4.3	1.05705	1.937
104.3	20.86	98793.9	9.8	1637.3	2.1	5.8	1.0921	1.404
54.8	10.96	97747.7	10.8	1225.7	2	6.6	1.12156	0.821
24.6	4.92	96337.4	14.5	852.5	4.5	13.3	1.3512	0.451
11.5	2.3	92767.5	9.6	609.9	9.2	23.5	1.52861	0.256
5.1	1.02	79355.4	4.9	522.7	1073.2	2567.1	13.6292	0.147
2.3	0.46	58436	2.5	506	1638.9	7200.6	24.6951	0.102
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.076
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.077

Tabelle A.28: $mgcd^{Pivot_{3+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2.6	1452.9	1	2	1.0139	3.187
450.1	90.02	99	2.6	1372.6	1	2.3	1.01552	2.912
403.9	80.78	99	2.3	1294.1	1	2	1.01035	2.642
350.7	70.14	99	2.7	1225	1	2.2	1.02041	2.333
295.6	59.12	98.9	2.6	1135.8	1	2.4	1.02051	2.033
253.4	50.68	98.8	2.1	1056.4	1	2.1	1.0088	1.742
198.7	39.74	98.7	2.5	972.1	1	2.2	1.01543	1.434
156.3	31.26	98.6	2.6	905.4	1	2.4	1.01833	1.139
97	19.4	98.7	3.1	785.4	1	2.6	1.01999	0.76
51.6	10.32	98.1	3.3	675.1	1	2.7	1.02266	0.437
29.5	5.9	97.1	3.4	605.4	1.1	2.5	1.02494	0.285
11	2.2	93.6	5.2	547.4	2.3	7.9	1.08221	0.149
5.4	1.08	82.9	3.7	516.8	6.3	20.8	1.13777	0.101
2.8	0.56	62.7	2.3	505.4	9.1	30.2	1.11694	0.063
1.8	0.36	57.3	1.7	502.2	2.6	13.4	1.0462	0.111
0.5	0.1000000	18.8	1	500	1	1	1	0.185

Tabelle A.29: $mgcd^{Pivot_{3+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.8	2172	1	2.3	1.01611	5.636
449	89.8	997.6	4	2128.8	1.1	2.9	1.03514	5.027
398.2	79.64	996	3.9	1959.1	1.1	2.9	1.03787	4.482
348.9	69.78	996.6	4.4	1813.4	1	2.6	1.02873	3.913
303.2	60.64	996.2	4.1	1633.6	1.1	3.1	1.03838	3.402
243.8	48.76	995.1	3.7	1429.1	1.1	3	1.03408	2.706
197.8	39.56	992.8	3.6	1261.5	1.1	2.6	1.02822	2.176
149	29.8	992.9	4.5	1131.8	1.3	3.2	1.03949	1.633
98	19.6	987.6	5	971.1	1.4	3.7	1.0485	1.091
48.4	9.68	978.9	5.2	779.1	1.6	4.1	1.06084	0.597
25.7	5.14	962.9	7.5	685.8	1.7	4.6	1.07437	0.364
8.6	1.72	828.9	6.1	546.7	2.9	8	1.11304	0.169
6.2	1.24	887.6	5.1	529.7	34.4	85.7	1.47631	0.136
2.6	0.52	624.3	2.6	505.3	46.4	244.6	1.91094	0.105
2	0.4	567.1	1.9	503.1	71.5	222.1	1.85669	0.099
1.1	0.22	257.8	1.4	501.4	5	13.9	1.05943	0.14

Tabelle A.30: $mgcd^{Pivot_{3+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	4.3	2591.8	1.1	2.9	1.03376	6.771
455.2	91.04	9985	4.8	2477.8	1	2.8	1.03273	6.171
397.3	79.46	9961.8	4.7	2315.7	1.1	3	1.03766	5.413
352.3	70.46	9977.6	4.9	2131.5	1	3	1.03167	4.839
301.7	60.34	9968.3	4.8	1970.5	1.1	3	1.03816	4.17
252.5	50.5	9945.4	4.9	1813.6	1	3	1.03667	3.529
197.8	39.56	9954.8	4.2	1570.2	1	3	1.0321	2.819
151.5	30.3	9926	5.3	1457.5	1.2	3.7	1.0483	2.192
99.9	19.98	9907.7	6.3	1243.6	1.3	3.9	1.05002	1.507
51.9	10.38	9677.3	8.8	974.1	1.7	5	1.07648	0.83
26.4	5.28	9739.2	9.8	775.2	2.4	6.1	1.12758	0.489
12.8	2.56	9122.1	9.4	612.5	3.5	13.2	1.26531	0.27
4.2	0.84	8115.2	4.1	515.5	405.3	964.3	4.7579	0.129
3.9	0.78	8317.4	3.7	512.6	384.9	1441.8	9.93894	0.121
1.7	0.34	4758.2	2	504.1	315.1	954.7	5.21762	0.156
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.109

Tabelle A.31: $mgcd^{Pivot_{3+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	5.3	3660.6	1.2	3.8	1.04718	8.907
452.1	90.42	99823.7	6.9	3716.9	1.5	4.2	1.05462	8.157
399.3	79.86	99763.5	5.3	3193.9	1.2	3.9	1.04737	7.278
352.9	70.58	99657.1	5.8	3046.3	1.1	4	1.04816	6.562
307.4	61.48	99466.2	6.7	2830.9	1.3	4	1.04013	5.666
251.8	50.36	99431.8	7.3	2547.6	1.2	4	1.04483	4.676
203.4	40.68	99511.4	8.6	2289.7	1.5	4.6	1.05813	3.781
150.7	30.14	99297.6	9	1968.6	1.4	4.3	1.05705	2.811
104.3	20.86	98793.9	9.8	1637.3	2.1	5.8	1.0921	1.974
54.8	10.96	97747.7	10.8	1225.7	2	6.6	1.12156	1.099
24.6	4.92	96337.4	14.5	852.5	4.5	13.3	1.3512	0.583
11.5	2.3	92767.5	9.6	609.9	9.2	23.5	1.52861	0.311
5.1	1.02	79355.4	4.9	522.7	1073.2	2567.1	13.6292	0.168
2.3	0.46	58436	2.5	506	1638.9	7200.6	24.6951	0.15
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.135
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.135

Tabelle A.32: $mgcd^{Pivot_{3+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2.6	1452.9	1	2	1.0139	2.371
450.1	90.02	99	2.6	1372.6	1	2.3	1.01552	2.179
403.9	80.78	99	2.3	1294.1	1	2	1.01035	1.983
350.7	70.14	99	2.7	1225	1	2.2	1.02041	1.758
295.6	59.12	98.9	2.6	1135.8	1	2.4	1.02051	1.54
253.4	50.68	98.8	2.1	1056.4	1	2.1	1.0088	1.337
198.7	39.74	98.7	2.5	972.1	1	2.2	1.01543	1.108
156.3	31.26	98.6	2.6	905.4	1	2.4	1.01833	0.885
97	19.4	98.7	3.1	785.4	1	2.6	1.01999	0.602
51.6	10.32	98.1	3.3	675.1	1	2.7	1.02266	0.36
29.5	5.9	97.1	3.4	605.4	1.1	2.5	1.02494	0.238
11	2.2	93.6	5.2	547.4	2.3	7.9	1.08221	0.131
5.4	1.08	82.9	3.7	516.8	6.3	20.8	1.13777	0.091
2.8	0.56	62.7	2.3	505.4	9.1	30.2	1.11694	0.063
1.8	0.36	57.3	1.7	502.2	2.6	13.4	1.0462	0.066
0.5	0.1000000	18.8	1	500	1	1	1	0.076

Tabelle A.33: $mgcd^{Pivot_{3+4}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	2.8	2172	1	2.3	1.01611	4.05
449	89.8	997.6	4	2128.8	1.1	2.9	1.03514	3.624
398.2	79.64	996	3.9	1959.1	1.1	2.9	1.03787	3.253
348.9	69.78	996.6	4.4	1813.4	1	2.6	1.02873	2.856
303.2	60.64	996.2	4.1	1633.6	1.1	3.1	1.03838	2.499
243.8	48.76	995.1	3.7	1429.1	1.1	3	1.03408	2
197.8	39.56	992.8	3.6	1261.5	1.1	2.6	1.02822	1.626
149	29.8	992.9	4.5	1131.8	1.3	3.2	1.03949	1.231
98	19.6	987.6	5	971.1	1.4	3.7	1.0485	0.84
48.4	9.68	978.9	5.2	779.1	1.6	4.1	1.06084	0.477
25.7	5.14	962.9	7.5	685.8	1.7	4.6	1.07437	0.298
8.6	1.72	828.9	6.1	546.7	2.9	8	1.11304	0.143
6.2	1.24	887.6	5.1	529.7	34.4	85.7	1.47631	0.12
2.6	0.52	624.3	2.6	505.3	46.4	244.6	1.91094	0.077
2	0.4	567.1	1.9	503.1	71.5	222.1	1.85669	0.074
1.1	0.22	257.8	1.4	501.4	5	13.9	1.05943	0.071

Tabelle A.34: $mgcd^{Pivot_{3+2}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	4.3	2591.8	1.1	2.9	1.03376	4.896
455.2	91.04	9985	4.8	2477.8	1	2.8	1.03273	4.469
397.3	79.46	9961.8	4.7	2315.7	1.1	3	1.03766	3.937
352.3	70.46	9977.6	4.9	2131.5	1	3	1.03167	3.53
301.7	60.34	9968.3	4.8	1970.5	1.1	3	1.03816	3.057
252.5	50.5	9945.4	4.9	1813.6	1	3	1.03667	2.598
197.8	39.56	9954.8	4.2	1570.2	1	3	1.0321	2.086
151.5	30.3	9926	5.3	1457.5	1.2	3.7	1.0483	1.634
99.9	19.98	9907.7	6.3	1243.6	1.3	3.9	1.05002	1.137
51.9	10.38	9677.3	8.8	974.1	1.7	5	1.07648	0.646
26.4	5.28	9739.2	9.8	775.2	2.4	6.1	1.12758	0.385
12.8	2.56	9122.1	9.4	612.5	3.5	13.2	1.26531	0.222
4.2	0.84	8115.2	4.1	515.5	405.3	964.3	4.7579	0.111
3.9	0.78	8317.4	3.7	512.6	384.9	1441.8	9.93894	0.103
1.7	0.34	4758.2	2	504.1	315.1	954.7	5.21762	0.086
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.066

Tabelle A.35: $mgcd^{Pivot_{3+4}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	5.3	3660.6	1.2	3.8	1.04718	6.181
452.1	90.42	99823.7	6.9	3716.9	1.5	4.2	1.05462	5.676
399.3	79.86	99763.5	5.3	3193.9	1.2	3.9	1.04737	5.082
352.9	70.58	99657.1	5.8	3046.3	1.1	4	1.04816	4.625
307.4	61.48	99466.2	6.7	2830.9	1.3	4	1.04013	4.001
251.8	50.36	99431.8	7.3	2547.6	1.2	4	1.04483	3.321
203.4	40.68	99511.4	8.6	2289.7	1.5	4.6	1.05813	2.701
150.7	30.14	99297.6	9	1968.6	1.4	4.3	1.05705	2.031
104.3	20.86	98793.9	9.8	1637.3	2.1	5.8	1.0921	1.446
54.8	10.96	97747.7	10.8	1225.7	2	6.6	1.12156	0.83
24.6	4.92	96337.4	14.5	852.5	4.5	13.3	1.3512	0.447
11.5	2.3	92767.5	9.6	609.9	9.2	23.5	1.52861	0.252
5.1	1.02	79355.4	4.9	522.7	1073.2	2567.1	13.6292	0.14
2.3	0.46	58436	2.5	506	1638.9	7200.6	24.6951	0.096
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.068
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.071

Tabelle A.36: $mgcd^{Pivot_{3+4}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2.1	1797.8	1.1	10	2.07954	3.66
450.1	90.02	99	1.1	1641.1	1.1	11.8	2.38834	3.326
403.9	80.78	99	1.1	1595.5	1.7	6.5	1.67133	3.198
350.7	70.14	99	1.1	1343.2	1.2	15.5	2.72163	2.417
295.6	59.12	98.9	1.1	1265.9	1.1	8.9	1.87384	2.157
253.4	50.68	98.8	1.2	1134	1.1	12	2.18818	1.834
198.7	39.74	98.7	1.5	1041.4	1.2	7.2	1.51997	1.514
156.3	31.26	98.6	1.4	882.6	1	18.1	2.38817	1.01
97	19.4	98.7	1.5	752.8	2.8	9.7	1.43185	0.674
51.6	10.32	98.1	1.9	625.4	4.2	13.9	1.42261	0.337
29.5	5.9	97.1	2	573.5	6.5	18.5	1.45057	0.218
11	2.2	93.6	2.5	525.9	7.4	34.5	1.38277	0.108
5.4	1.08	82.9	2.1	508.5	14.9	62.4	1.39921	0.07
2.8	0.56	62.7	2	503.7	11.3	37.3	1.16994	0.059
1.8	0.36	57.3	1.7	502	2.3	12.5	1.03446	0.052
0.5	0.1000000	18.8	1	500	1	1	1	0.046

Tabelle A.37: $mgcd^{Pivot_{4+1}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	3	2176.2	3.8	40.9	6.36752	4.282
449	89.8	997.6	1.7	2181.5	1.1	22.4	3.09824	5.154
398.2	79.64	996	1.7	2032	3.4	21.5	2.71619	4.625
348.9	69.78	996.6	1.7	1662.9	2.5	31.8	4.30826	3.363
303.2	60.64	996.2	2.1	1715.1	6	21.8	2.47875	3.497
243.8	48.76	995.1	2	1355.2	24.4	74.3	4.87205	2.641
197.8	39.56	992.8	1.8	1224.4	2	36.3	3.68376	2.076
149	29.8	992.9	2.1	1036.7	4	25.6	2.67879	1.441
98	19.6	987.6	2	832	24.9	80	4.40938	0.862
48.4	9.68	978.9	2.5	668.7	13.4	81.1	3.23239	0.438
25.7	5.14	962.9	2.9	583.9	17.1	122.6	3.22418	0.24
8.6	1.72	828.9	2.9	524.1	43.3	221.3	2.63499	0.111
6.2	1.24	887.6	2.5	512.2	112.9	431.4	4.45588	0.099
2.6	0.52	624.3	2.2	503.8	49.4	277.7	2.01727	0.067
2	0.4	567.1	1.6	502.2	67.6	406.5	3.12804	0.055
1.1	0.22	257.8	1.2	500.8	29.2	105.7	1.49141	0.054

Tabelle A.38: $mgcd^{Pivot_{4+1}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	3.4	2300.9	214.7	3327.2	394	4.136
455.2	91.04	9985	2.4	2630.2	86.3	427.3	27.3653	6.477
397.3	79.46	9961.8	2.7	2426.6	12.6	89.4	6.49089	5.747
352.3	70.46	9977.6	2.4	2208.1	5.7	57.8	4.76736	4.997
301.7	60.34	9968.3	2.4	1926.7	11.9	46.3	4.22972	4.151
252.5	50.5	9945.4	2.6	1731.5	82.7	237.6	11.484	3.591
197.8	39.56	9954.8	2.7	1505.3	20.2	57.4	3.73553	2.913
151.5	30.3	9926	2.4	1115.7	421.9	1136.2	39.0835	1.753
99.9	19.98	9907.7	2.8	958.1	62.1	207.5	8.52374	1.253
51.9	10.38	9677.3	3.1	727.1	247	859.6	26.649	0.632
26.4	5.28	9739.2	3.5	614.1	198	467.9	7.9531	0.342
12.8	2.56	9122.1	2.9	538.1	363.7	2747.7	32.3657	0.178
4.2	0.84	8115.2	2.3	506.8	1097.6	5013	25.7624	0.084
3.9	0.78	8317.4	2.4	506.5	1276.4	4525.4	29.2274	0.086
1.7	0.34	4758.2	1.6	502.7	396.5	1395.1	7.89238	0.057
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.052

Tabelle A.39: $mgcd^{Pivot_{4+1}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	3.2	2030.8	2211.6	27722.3	3789.54	3.695
452.1	90.42	99823.7	3.6	3029.8	79.4	286.7	18.0369	7.226
399.3	79.86	99763.5	3.2	2875.8	51.2	247.8	14.0596	6.753
352.9	70.58	99657.1	2.9	2473	57.5	4458.5	311.4	6.144
307.4	61.48	99466.2	3.1	2167.1	1587.9	5320.6	224.582	5.226
251.8	50.36	99431.8	3.6	2000.4	302.5	3172.3	179.757	4.388
203.4	40.68	99511.4	3.3	1700.4	719.5	1538.6	51.8015	3.379
150.7	30.14	99297.6	3.5	1408.6	39.2	218	10.5578	2.547
104.3	20.86	98793.9	3.9	1099.4	607	2571.5	76.3451	1.665
54.8	10.96	97747.7	3.9	755.6	4474.1	11865.4	273.597	0.799
24.6	4.92	96337.4	3.4	595.9	3461.1	16315	314.158	0.371
11.5	2.3	92767.5	3.1	539.2	3657.2	24600.4	249.579	0.193
5.1	1.02	79355.4	2.3	509.1	12918.5	43541.7	267.486	0.109
2.3	0.46	58436	1.8	503.1	5104.3	29077.6	141.391	0.07
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.052
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.049

Tabelle A.40: $mgcd^{Pivot_{4+1}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2.3	1943.9	1	6.8	1.64206	4.888
450.1	90.02	99	1.1	1671.6	1.6	10.5	2.14537	3.962
403.9	80.78	99	1.1	1598.6	1.7	6.5	1.65126	3.799
350.7	70.14	99	1.1	1358.3	1.2	15	2.63557	2.906
295.6	59.12	98.9	1.1	1280.3	1.1	6.8	1.69312	2.622
253.4	50.68	98.8	1.2	1137.2	1.1	11.8	2.17112	2.153
198.7	39.74	98.7	1.5	1047.8	1	6.9	1.48673	1.804
156.3	31.26	98.6	1.4	885.6	1	17.7	2.35772	1.184
97	19.4	98.7	1.5	757.3	2.7	10	1.42321	0.783
51.6	10.32	98.1	1.9	625.4	4.2	13.9	1.42261	0.375
29.5	5.9	97.1	2	574.6	4.5	15.8	1.37696	0.249
11	2.2	93.6	2.6	525.6	10.3	37.7	1.37881	0.122
5.4	1.08	82.9	2	508.9	14.1	60.1	1.35862	0.075
2.8	0.56	62.7	2	503.7	10.2	36	1.16835	0.061
1.8	0.36	57.3	1.7	502	2.3	12.5	1.03446	0.057
0.5	0.1000000	18.8	1	500	1	1	1	0.044

Tabelle A.41: $mgcd^{Pivot_{4+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	4.5	2840.7	2.7	22.6	3.73348	6.63
449	89.8	997.6	1.7	2173	1.1	22.4	3.06208	6.388
398.2	79.64	996	1.7	2057.5	2.9	14.7	2.25055	5.698
348.9	69.78	996.6	1.7	1698.3	3.7	30.2	3.94359	4.077
303.2	60.64	996.2	2.1	1712.2	6	23	2.55046	4.149
243.8	48.76	995.1	2	1356.4	24.4	74.3	4.84606	3.161
197.8	39.56	992.8	1.8	1224.4	2	36.3	3.68376	2.423
149	29.8	992.9	2.1	1042.8	4.9	27.8	2.73677	1.677
98	19.6	987.6	2	835.8	25.1	80.4	4.36791	1.015
48.4	9.68	978.9	2.4	671.5	21.7	90.7	3.24974	0.509
25.7	5.14	962.9	3	593.8	13.3	83.1	2.2585	0.294
8.6	1.72	828.9	2.9	524.6	45.9	225.2	2.63629	0.121
6.2	1.24	887.6	2.7	513.5	121.4	406.2	4.45141	0.104
2.6	0.52	624.3	2.2	503.8	49.4	277.7	2.01727	0.067
2	0.4	567.1	1.7	502.3	65.2	391.3	3.10293	0.059
1.1	0.22	257.8	1.2	500.8	29.2	105.7	1.49141	0.052

Tabelle A.42: $mgcd^{Pivot_{4+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	4.8	3249.5	42	1358.3	109.625	7.546
455.2	91.04	9985	2.6	2861.9	5.8	27	2.97509	8.326
397.3	79.46	9961.8	2.6	2475.4	10.5	81.4	5.86063	7.075
352.3	70.46	9977.6	2.5	2265.6	6.5	48	4.13886	6.205
301.7	60.34	9968.3	2.4	1924.9	11.7	47.6	4.37753	4.904
252.5	50.5	9945.4	2.6	1752.9	81.2	233.9	11.5075	4.402
197.8	39.56	9954.8	2.8	1529.8	18.6	55.9	3.59001	3.531
151.5	30.3	9926	2.4	1122.4	383.6	1104	38.5267	2.09
99.9	19.98	9907.7	2.8	959.1	62.1	207.5	8.52622	1.445
51.9	10.38	9677.3	3.1	727.5	247	859.4	26.5649	0.738
26.4	5.28	9739.2	3.5	610.9	194.8	470.2	7.99198	0.388
12.8	2.56	9122.1	3.5	544	257.4	977.3	10.4399	0.207
4.2	0.84	8115.2	2.6	507.5	885.2	3623.4	19.1793	0.091
3.9	0.78	8317.4	2.6	507.3	1242.5	4207.8	25.9923	0.092
1.7	0.34	4758.2	1.8	503.6	391.6	1322	7.45294	0.065
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.056

Tabelle A.43: $mgcd^{Pivot_{4+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	5.3	3872.3	174	808.4	90.5167	10.406
452.1	90.42	99823.7	3.4	3200.2	7.7	124.4	10.4427	9.084
399.3	79.86	99763.5	3.5	3008.4	48.9	233.9	12.2408	8.699
352.9	70.58	99657.1	3	2558.6	28.5	149	10.7433	7.883
307.4	61.48	99466.2	3.1	2206.6	1587.9	5305.8	221.607	6.421
251.8	50.36	99431.8	3.6	2039.3	280.2	3103.7	173.091	5.337
203.4	40.68	99511.4	3.4	1737.9	693.8	1485.1	49.2715	4.179
150.7	30.14	99297.6	3.7	1420.8	21.6	98.5	6.2859	3.081
104.3	20.86	98793.9	3.8	1122.3	602.4	2565.8	74.7681	1.986
54.8	10.96	97747.7	4.1	785.1	4453.6	10640.3	233.297	0.958
24.6	4.92	96337.4	3.4	597.6	3464.4	16316.7	313.683	0.417
11.5	2.3	92767.5	3.7	545.6	3785.4	21961.8	217.395	0.217
5.1	1.02	79355.4	2.3	509.1	11985.4	41759.5	253.882	0.118
2.3	0.46	58436	1.8	503.1	5104.3	29077.6	141.391	0.077
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.053
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.053

Tabelle A.44: $mgcd^{Pivot_{4+2}}_{BestRemainder}(k = 1)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2.1	1917.9	1	8.3	1.76255	8.407
450.1	90.02	99	1.1	1663.7	1.1	9.3	2.03961	6.617
403.9	80.78	99	1.1	1615.4	1.7	6.2	1.60635	6.357
350.7	70.14	99	1.1	1360.1	1.2	12.2	2.29888	4.769
295.6	59.12	98.9	1.1	1272.5	1.1	8.6	1.84244	4.07
253.4	50.68	98.8	1.2	1139	1.1	11.8	2.16488	3.568
198.7	39.74	98.7	1.5	1045.8	1.2	6.7	1.47485	2.892
156.3	31.26	98.6	1.4	890.3	1	18.4	2.33157	1.843
97	19.4	98.7	1.5	760.2	2.5	9.7	1.41805	1.192
51.6	10.32	98.1	1.9	625.4	4.2	13.9	1.42261	0.508
29.5	5.9	97.1	2	575.2	4.4	14.4	1.34336	0.335
11	2.2	93.6	2.5	527.2	7.4	33.8	1.37709	0.143
5.4	1.08	82.9	2.1	508.5	13.3	55.9	1.36323	0.086
2.8	0.56	62.7	2	503.8	9.8	34.9	1.15661	0.061
1.8	0.36	57.3	1.7	502	2.3	12.5	1.03446	0.056
0.5	0.1000000	18.8	1	500	1	1	1	0.046

Tabelle A.45: $mgcd^{Pivot_{4+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	3.2	2670.2	1.8	17.1	2.99397	11.937
449	89.8	997.6	1.7	2211.8	1.2	22.3	3.02722	11.629
398.2	79.64	996	1.7	2069.2	3.4	14.2	2.22883	10.217
348.9	69.78	996.6	1.7	1684.4	2.5	29.2	4.0076	6.689
303.2	60.64	996.2	2.1	1706.4	6	22	2.43724	7.111
243.8	48.76	995.1	2	1376.4	24.4	74.3	4.75182	5.421
197.8	39.56	992.8	1.9	1243.3	1.9	35.2	3.54589	4.103
149	29.8	992.9	2.1	1041.4	5	27.1	2.68648	2.665
98	19.6	987.6	2	830.7	24.9	80	4.39063	1.485
48.4	9.68	978.9	2.4	671.3	21.7	95.8	3.37986	0.721
25.7	5.14	962.9	2.9	585.3	13.2	102.1	2.76439	0.361
8.6	1.72	828.9	2.9	526.1	36.5	141.5	2.00323	0.152
6.2	1.24	887.6	2.6	513.6	110.5	386.3	4.25467	0.125
2.6	0.52	624.3	2.2	503.8	49.4	277.7	2.01727	0.072
2	0.4	567.1	1.7	502.4	58	354.2	2.88953	0.068
1.1	0.22	257.8	1.2	500.8	29.2	105.7	1.49141	0.056

Tabelle A.46: $mgcd^{Pivot_{4+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	4.1	3634.8	1.6	32.9	5.48688	14.534
455.2	91.04	9985	2.6	2803.1	84.5	411.5	24.5054	14.296
397.3	79.46	9961.8	2.5	2403.3	14.1	94.1	6.82778	12.356
352.3	70.46	9977.6	2.4	2214.3	7.1	53.8	4.46981	10.863
301.7	60.34	9968.3	2.5	1949.3	11.4	45.5	4.15539	8.488
252.5	50.5	9945.4	2.6	1755.6	81.1	234.8	11.1752	7.495
197.8	39.56	9954.8	2.7	1497.2	19.8	62	4.04068	5.837
151.5	30.3	9926	2.6	1163.4	383	1081	35.5285	3.627
99.9	19.98	9907.7	2.8	965	62.1	207.5	8.47378	2.287
51.9	10.38	9677.3	3.1	728.5	236	838.2	26.4246	1.087
26.4	5.28	9739.2	3.5	615.7	194.8	465.9	7.93633	0.549
12.8	2.56	9122.1	3.6	551.4	252.9	1388.5	14.765	0.273
4.2	0.84	8115.2	2.6	508.3	766.7	3551.1	19.8428	0.108
3.9	0.78	8317.4	2.6	507.3	1174.3	4267.1	26.9728	0.103
1.7	0.34	4758.2	1.6	502.9	339.3	1072.8	6.49135	0.068
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.055

Tabelle A.47: $mgcd^{Pivot_{4+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	5.3	4120.1	1.5	32.1	4.19216	18.943
452.1	90.42	99823.7	3.3	3237.1	7.2	79.1	6.63375	17.923
399.3	79.86	99763.5	3.2	2899.1	10.2	83.4	7.14149	14.804
352.9	70.58	99657.1	2.9	2500.4	31	214.4	15.4598	13.825
307.4	61.48	99466.2	3.1	2211.9	1587.5	5305	220.907	11.543
251.8	50.36	99431.8	3.8	2034.1	274.9	3081.5	173.24	9.578
203.4	40.68	99511.4	3.3	1709.9	712.7	1538.1	51.3723	7.228
150.7	30.14	99297.6	3.6	1418.7	187.8	451.4	16.5625	5.379
104.3	20.86	98793.9	3.8	1121.5	602.2	2565.3	74.851	3.264
54.8	10.96	97747.7	3.8	749.4	5547.7	13788.3	320.853	1.383
24.6	4.92	96337.4	3.4	595.9	3461.1	16315	314.158	0.581
11.5	2.3	92767.5	3.7	549.3	3676.1	21457.9	204.965	0.289
5.1	1.02	79355.4	2.3	509.1	11985.4	41759.5	253.882	0.141
2.3	0.46	58436	1.8	503.1	5104.3	29077.6	141.391	0.081
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.05
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.054

Tabelle A.48: $mgcd^{Pivot_{4+2}}_{BestRemainder}(k = 2)$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99	2	1926.4	1	5.5	1.45276	4.64
450.1	90.02	99	1.1	1711.5	1.1	7.2	1.71475	3.618
403.9	80.78	99	1.1	1618.6	2.1	6.9	1.64982	3.361
350.7	70.14	99	1.1	1418.7	1.2	10.1	2.00141	2.757
295.6	59.12	98.9	1.1	1300.2	1.1	6.7	1.6419	2.35
253.4	50.68	98.8	1.2	1149.6	1.1	11.6	2.06889	1.914
198.7	39.74	98.7	1.5	1051.2	1	6.3	1.42237	1.581
156.3	31.26	98.6	1.4	892.5	1	18	2.30174	1.061
97	19.4	98.7	1.5	763.2	2.5	9.6	1.39072	0.706
51.6	10.32	98.1	1.9	625.4	4.2	13.9	1.42261	0.339
29.5	5.9	97.1	2.1	578	3.6	14.4	1.31453	0.231
11	2.2	93.6	2.6	527.5	9.8	33.3	1.3545	0.11
5.4	1.08	82.9	2	509.4	13.1	56	1.32587	0.075
2.8	0.56	62.7	2	503.8	11.9	38.5	1.16931	0.057
1.8	0.36	57.3	1.6	502	2.3	13.1	1.03805	0.05
0.5	0.1000000	18.8	1	500	1	1	1	0.044

Tabelle A.49: $mgcd^{Pivot_{4+3}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	997.6	3.8	2941	1.7	22.5	4.51224	8.711
449	89.8	997.6	1.7	2207.6	1.5	21.8	2.92594	5.631
398.2	79.64	996	1.7	2097.8	2.9	12.4	1.92273	5.082
348.9	69.78	996.6	1.7	1715.3	1.5	26.8	3.76191	3.557
303.2	60.64	996.2	2.1	1728.7	6	21.8	2.38133	3.614
243.8	48.76	995.1	2	1381	24.4	74.3	4.72114	2.732
197.8	39.56	992.8	1.9	1246.1	1.9	35.2	3.53013	2.148
149	29.8	992.9	2.2	1039.5	4.5	33.8	2.91159	1.472
98	19.6	987.6	2	836.8	25.1	80.4	4.32935	0.896
48.4	9.68	978.9	2.4	671.9	21.7	95.7	3.32936	0.452
25.7	5.14	962.9	3	593.4	16.4	103.2	2.57297	0.27
8.6	1.72	828.9	2.9	525.9	40.8	148	2.09888	0.115
6.2	1.24	887.6	2.6	513.3	100.2	380.6	4.22989	0.102
2.6	0.52	624.3	2.2	503.7	51	346.7	2.37066	0.063
2	0.4	567.1	1.9	502.8	60.3	291	2.6078	0.061
1.1	0.22	257.8	1.2	500.8	29.2	105.7	1.49141	0.052

Tabelle A.50: $mgcd^{Pivot_{4+3}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 1000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	9978.8	9	5465.8	6.8	17.7	2.62055	14.569
455.2	91.04	9985	2.6	2886.5	4.3	23.2	2.65484	7.148
397.3	79.46	9961.8	2.4	2494	6.1	38.1	3.59876	6.295
352.3	70.46	9977.6	2.4	2264.2	5.2	44.2	3.87863	5.388
301.7	60.34	9968.3	2.5	1931.7	11.3	44.4	4.18714	4.257
252.5	50.5	9945.4	2.6	1755.9	82.6	234.8	11.5209	3.753
197.8	39.56	9954.8	2.7	1504.7	19.8	63.2	4.05775	2.996
151.5	30.3	9926	2.6	1166.7	421.3	1111.3	35.4697	1.884
99.9	19.98	9907.7	2.8	965.2	62.1	207.5	8.45856	1.266
51.9	10.38	9677.3	3.1	727.2	247	859.4	26.5796	0.637
26.4	5.28	9739.2	3.5	610.8	225.1	658.3	11.1509	0.344
12.8	2.56	9122.1	3.7	549.3	182.7	753.4	7.8276	0.194
4.2	0.84	8115.2	2.6	508.5	785.2	3597.2	20.753	0.091
3.9	0.78	8317.4	2.8	508.8	1208.7	3835.3	24.4448	0.087
1.7	0.34	4758.2	1.7	503.4	320	1012.4	5.89213	0.059
1	0.2	4587.9	1.2	500.4	77.8	390.7	2.57674	0.052

Tabelle A.51: $mgcd^{Pivot_{4+3}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 10000$)

$L_0(v)$	$\varrho(v)$	$L_\infty(v)$	$L_0(x)$	$L_0(T)$	$L_\infty(x)$	$L_\infty(T)$	$\Delta_\infty(T)$	t
500	100	99891.8	5.7	3559.8	61.3	619.8	89.4083	112.135
452.1	90.42	99823.7	4.6	3409.2	7.5	63.6	5.5513	8.481
399.3	79.86	99763.5	3.4	3033.8	8.3	69.3	5.51635	7.613
352.9	70.58	99657.1	2.9	2548.4	31	212.8	14.8138	6.506
307.4	61.48	99466.2	3.1	2215.6	1587.9	5305.8	220.68	5.445
251.8	50.36	99431.8	4	2224.3	49.1	143.5	6.55595	4.769
203.4	40.68	99511.4	3.4	1739.3	695.4	1485.2	49.2155	3.576
150.7	30.14	99297.6	3.6	1443.2	11.8	80.1	5.09479	2.629
104.3	20.86	98793.9	3.9	1126.6	405.3	1115.1	33.7382	1.721
54.8	10.96	97747.7	3.9	789.6	4550.8	9822.9	217.763	0.822
24.6	4.92	96337.4	3.4	597.4	3464.4	16316.7	313.056	0.37
11.5	2.3	92767.5	3.7	546.7	3660.5	21435.3	205.269	0.201
5.1	1.02	79355.4	2.3	509.1	12918.5	43541.7	267.486	0.11
2.3	0.46	58436	1.9	503.6	5198.2	28116.5	136.795	0.074
0.8	0.16	47270.4	1.1	500.2	1802.8	9816.3	33.974	0.048
0.9	0.18	39882.1	1.2	500.4	2179.4	8483	28.3803	0.049

Tabelle A.52: $mgcd^{Pivot_{4+3}}_{BestRemainder}$ Laufzeitergebnisse ($L_\infty(A) < 100000$)

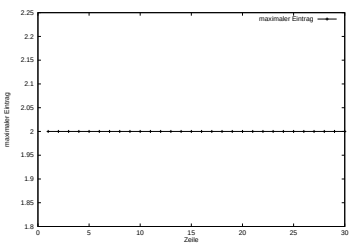
Anhang B

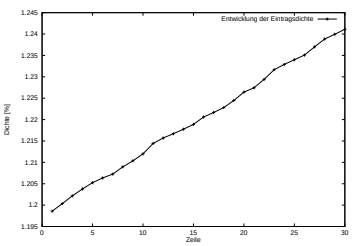
Ergebnisse: nichtmodulare, zeilenweise HNF–Berechnung

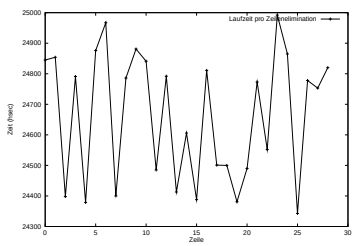
In diesem Kapitel haben wir die Laufzeitergebnisse der nichtmodularen HNF–Algorithmen, die sich einer zeilenweisen Vorgehensweise bedienen, für die Beispielmatrizen $BSP_{Jacobson}$ und BSP_{Neis} in Form von Grafiken zusammengestellt. Unser Ziel ist es, den Einfluß der *mgcd*–Berechnung auf die Entwicklung des maximalen Eintrags, der Eintragsdichte und der Laufzeit pro Zeilenelimination zu ermitteln. Aus diesem Grund wurde als Kernalgorithmus HNF_{Z1} fixiert. Wir erhalten die folgenden Ergebnisse:

B.1 $BSP_{Jacobson}$

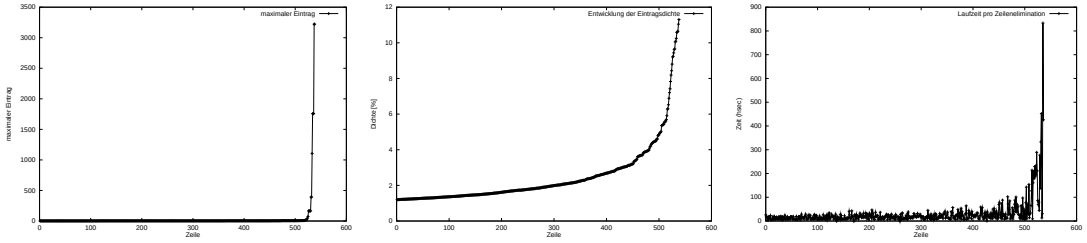
mgcd–Kennzahl: 1



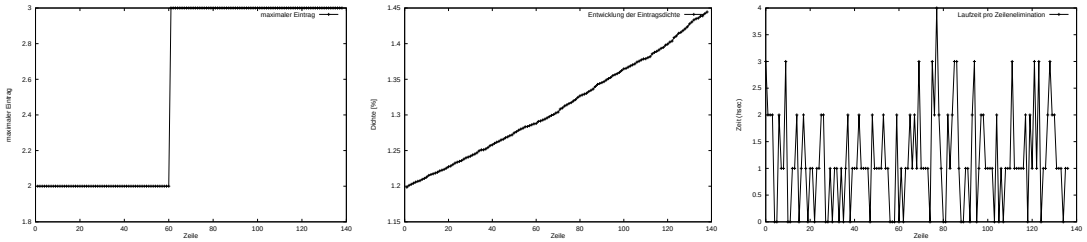




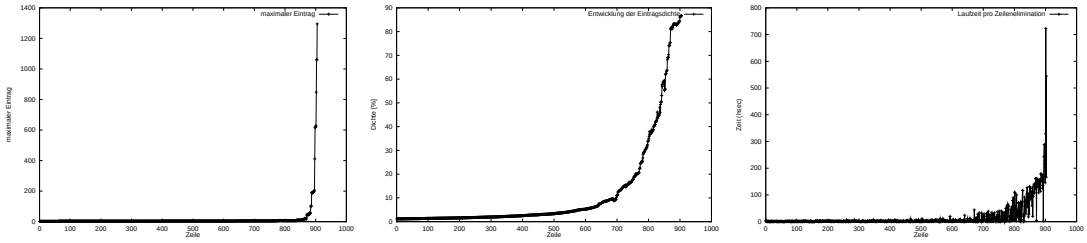
mgcd–Kennzahl: 2



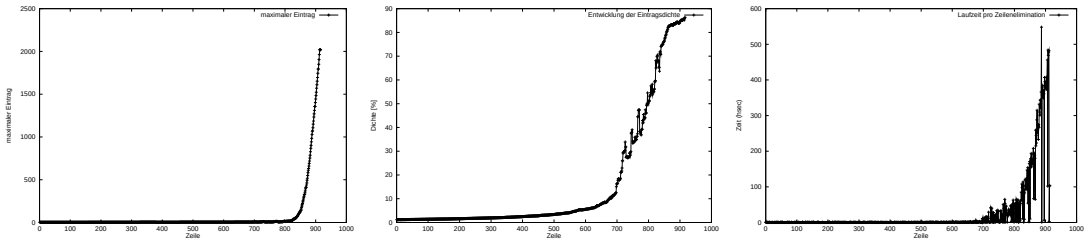
mgcd–Kennzahl: 3



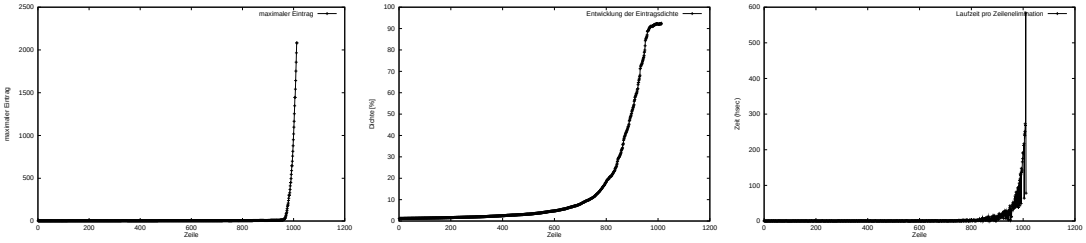
mgcd–Kennzahl: 4



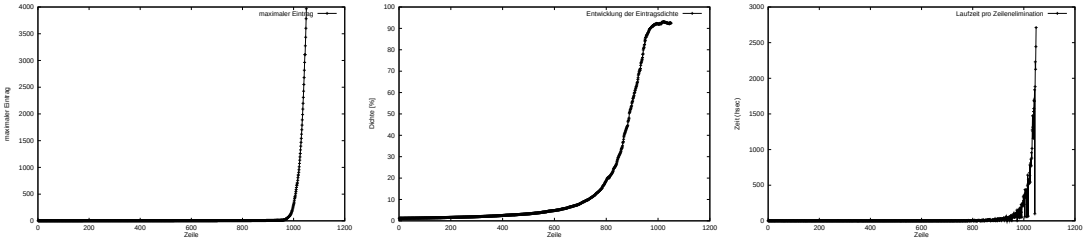
mgcd–Kennzahl: 5



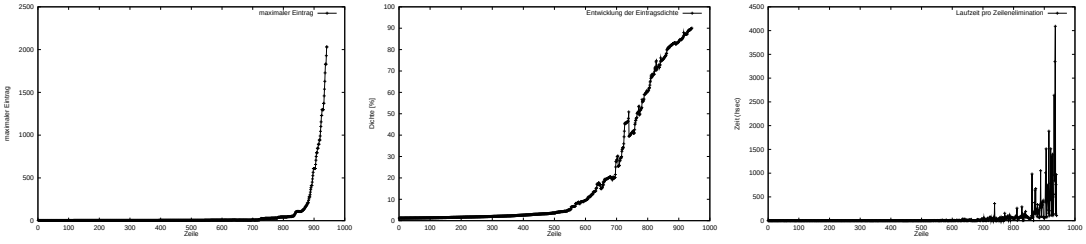
mgcd–Kennzahl: 6



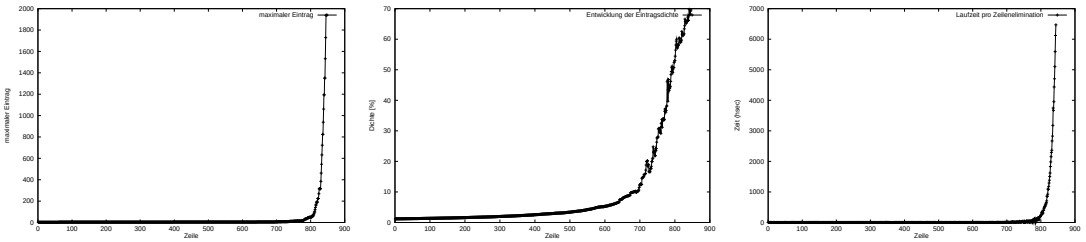
mgcd–Kennzahl: 7



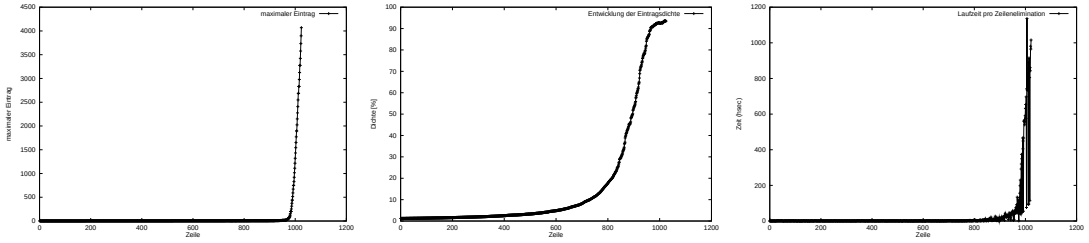
mgcd–Kennzahl: 8



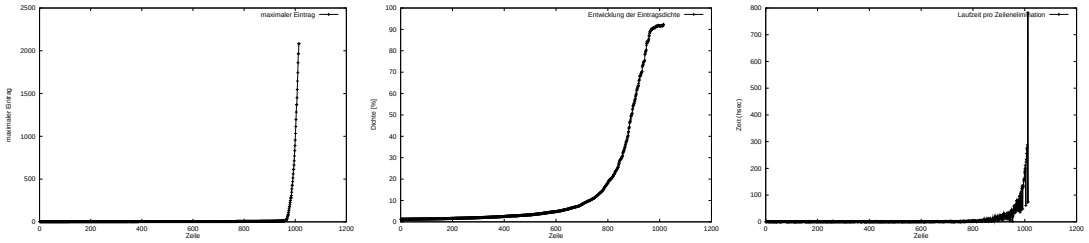
mgcd–Kennzahl: 9



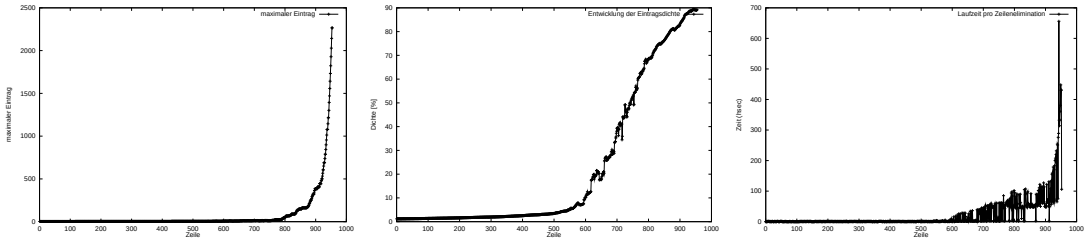
mgcd–Kennzahl: 10



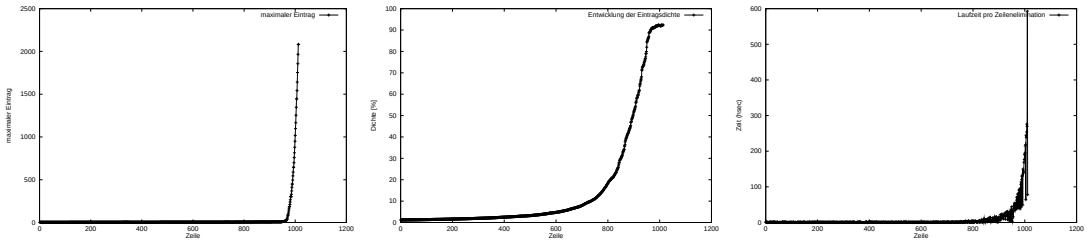
mgcd–Kennzahl: 11



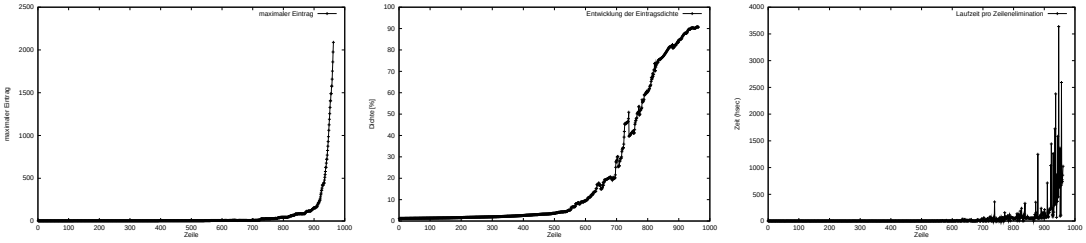
mgcd–Kennzahl: 12



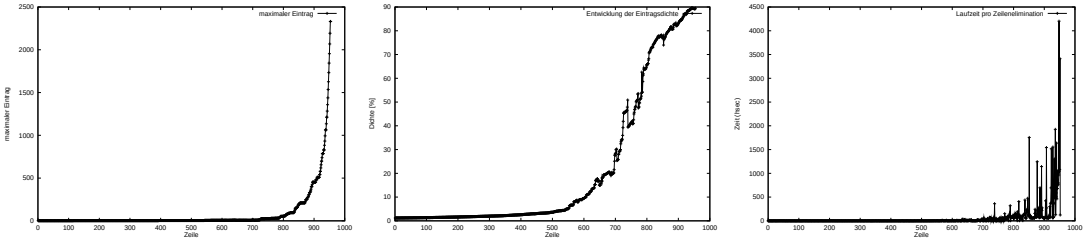
mgcd–Kennzahl: 13



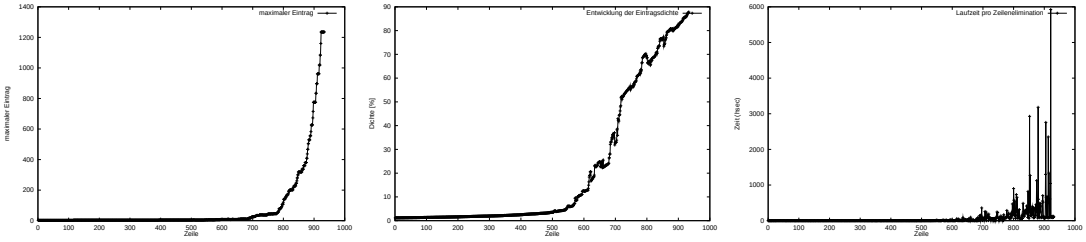
mgcd–Kennzahl: 14



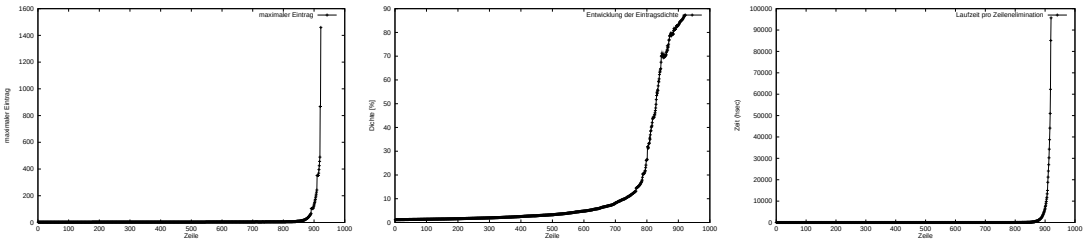
mgcd–Kennzahl: 15



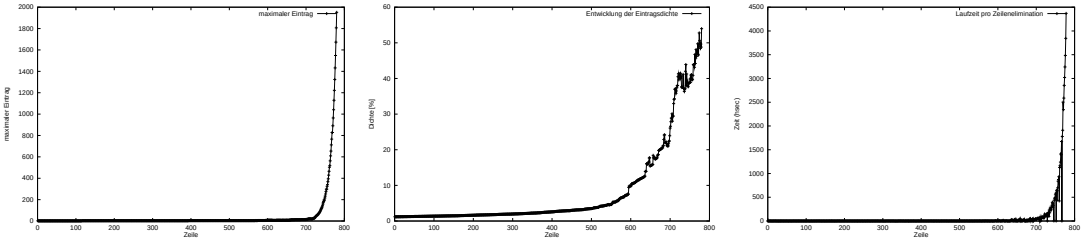
mgcd–Kennzahl: 16



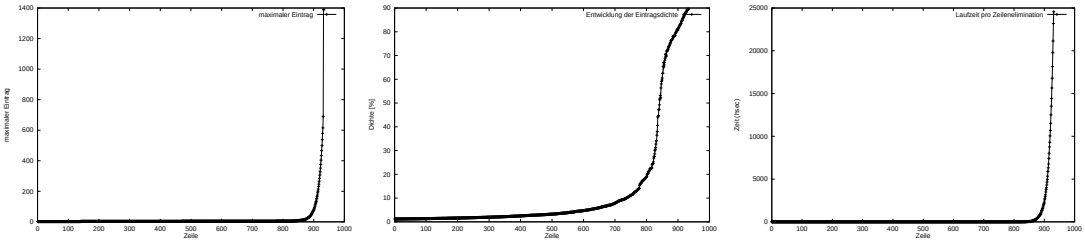
mgcd–Kennzahl: 17



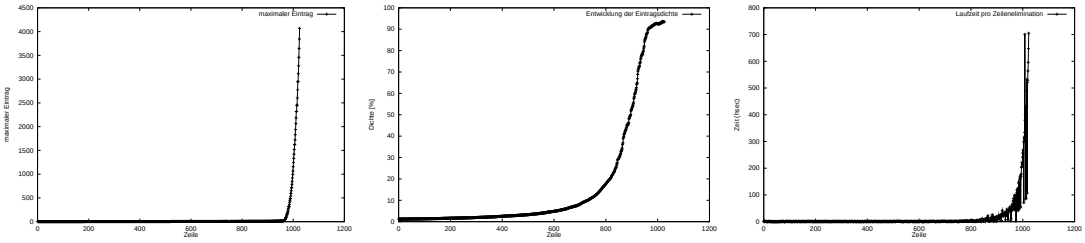
mgcd–Kennzahl: 18



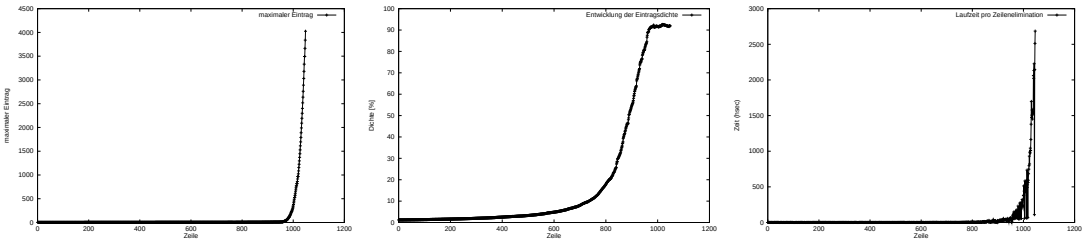
mgcd–Kennzahl: 19



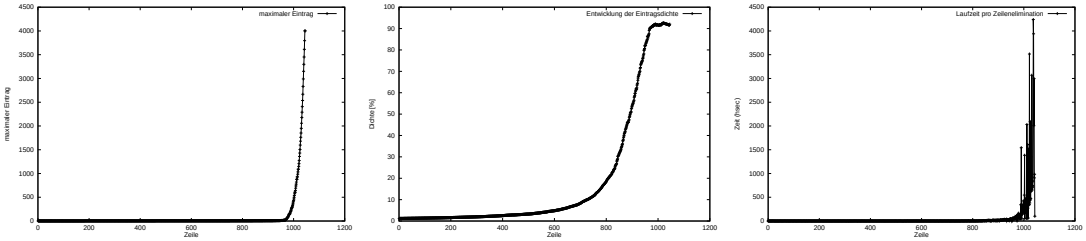
mgcd–Kennzahl: 20



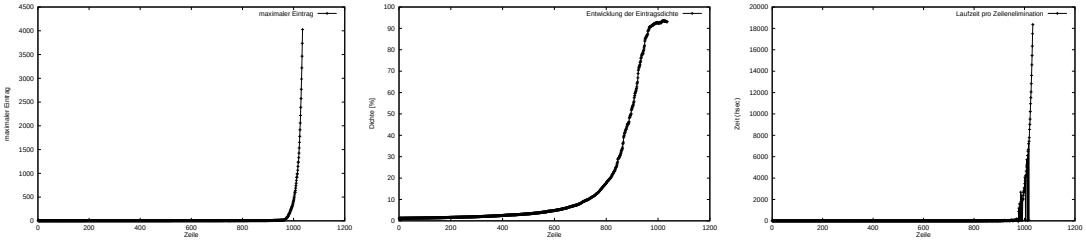
mgcd–Kennzahl: 21



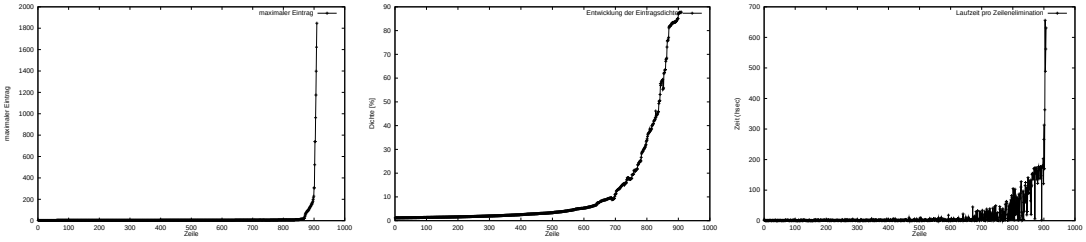
mgcd–Kennzahl: 22



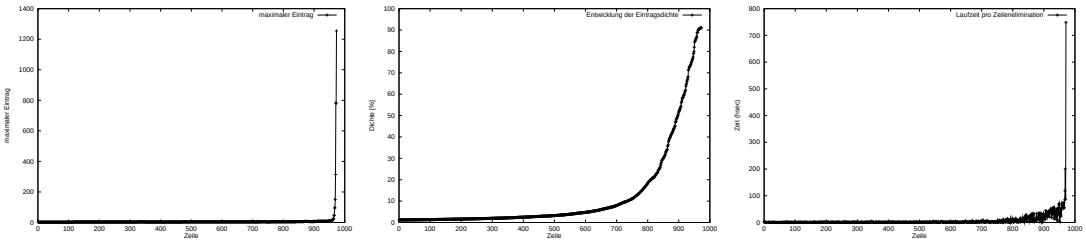
mgcd–Kennzahl: 23



mgcd–Kennzahl: 24

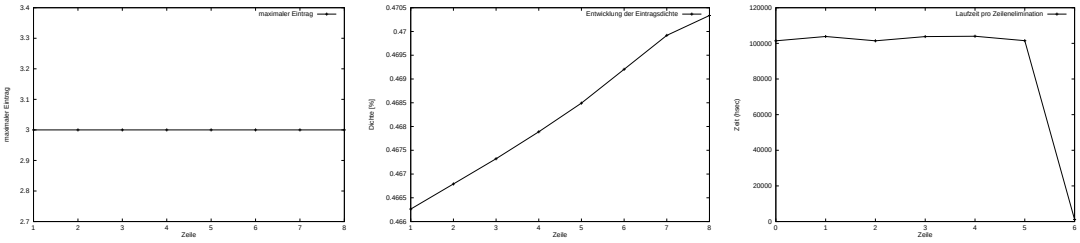


mgcd–Kennzahl: 25

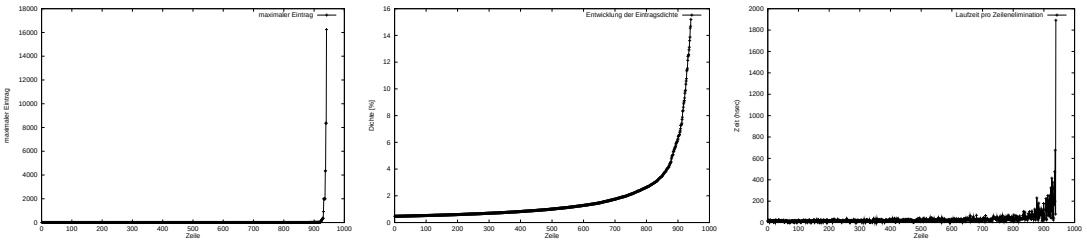


B.2 BSP_{Neis}

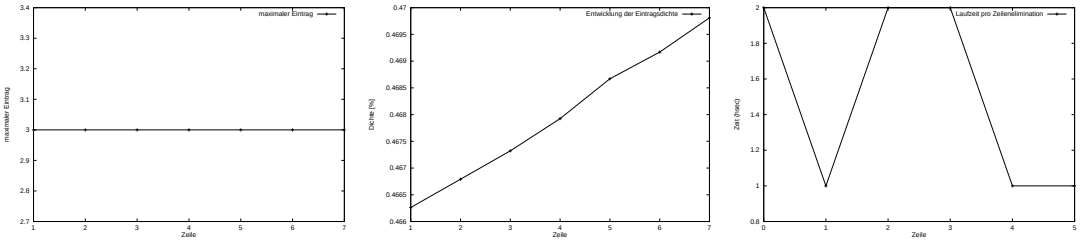
mgcd–Kennzahl: 1



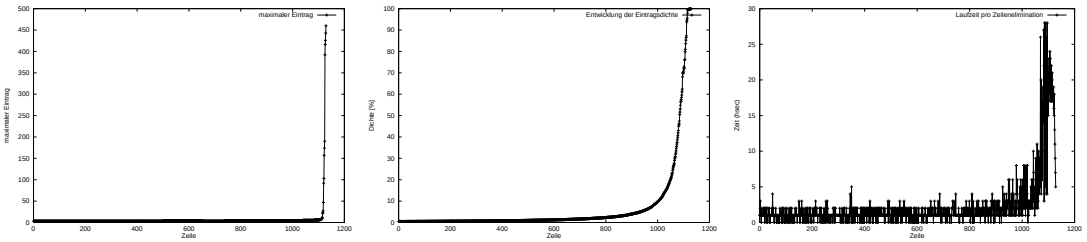
mgcd–Kennzahl: 2



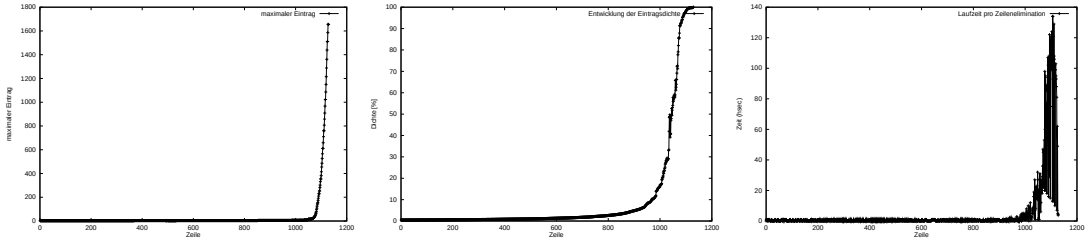
mgcd–Kennzahl: 3



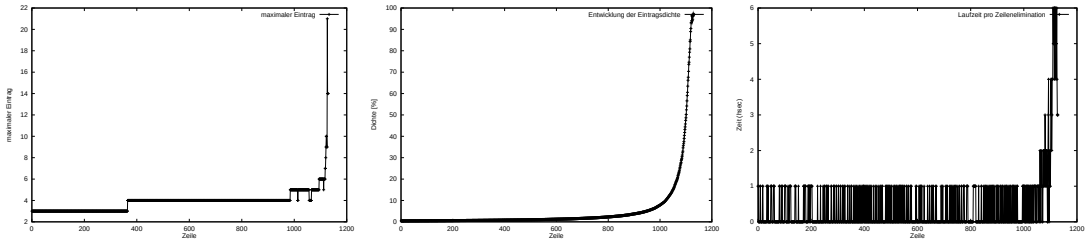
mgcd–Kennzahl: 4



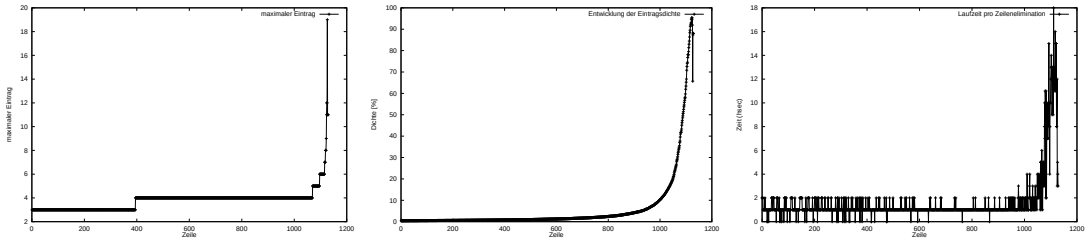
mgcd–Kennzahl: 5



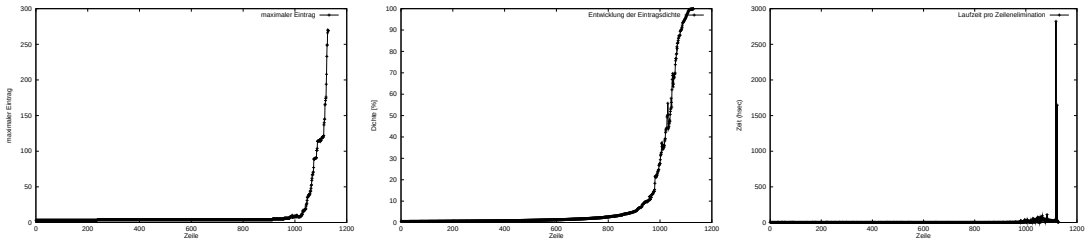
mgcd–Kennzahl: 6



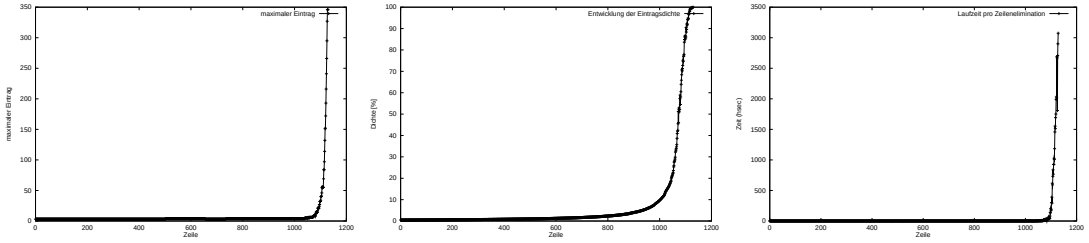
mgcd–Kennzahl: 7



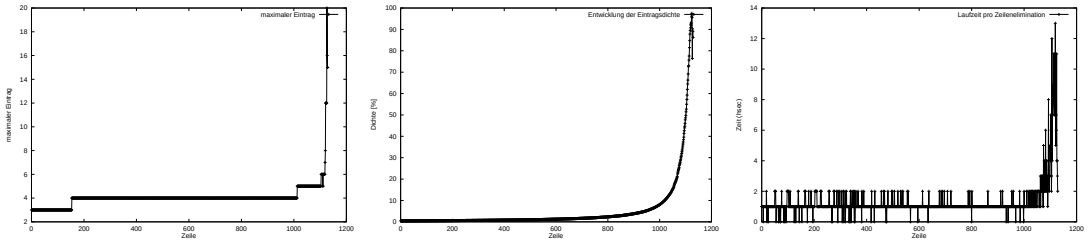
mgcd–Kennzahl: 8



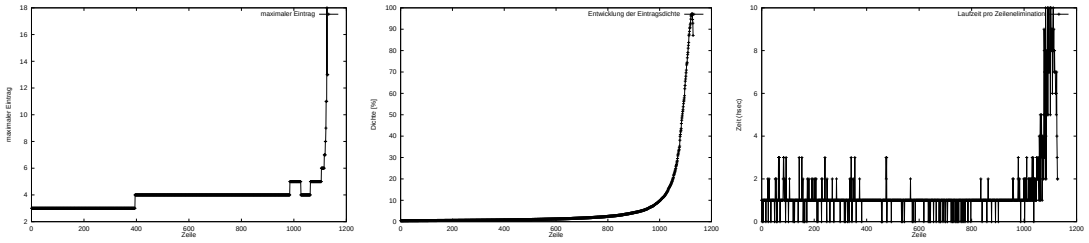
mgcd–Kennzahl: 9



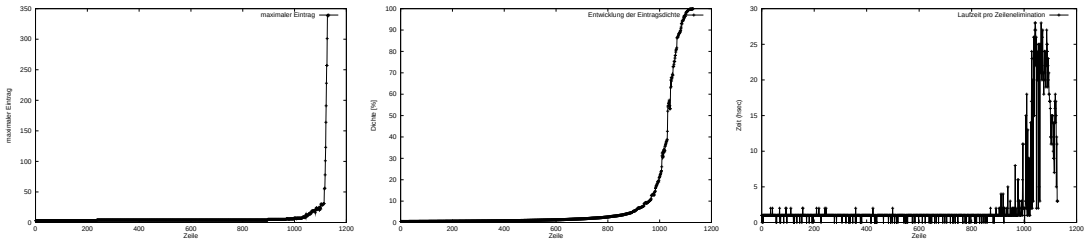
mgcd–Kennzahl: 10



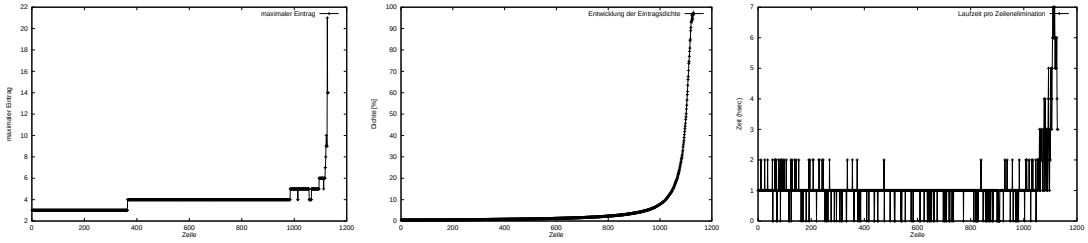
mgcd–Kennzahl: 11



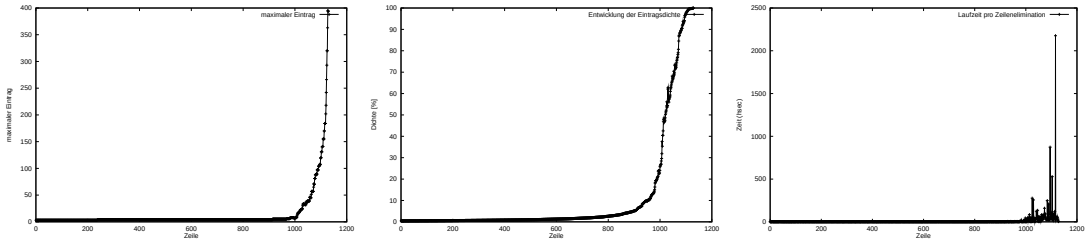
mgcd–Kennzahl: 12



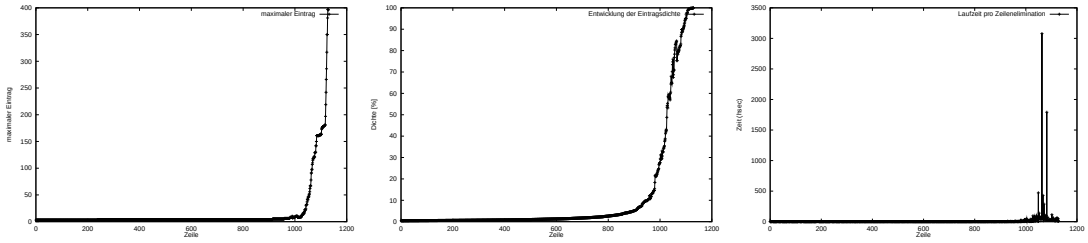
mgcd–Kennzahl: 13



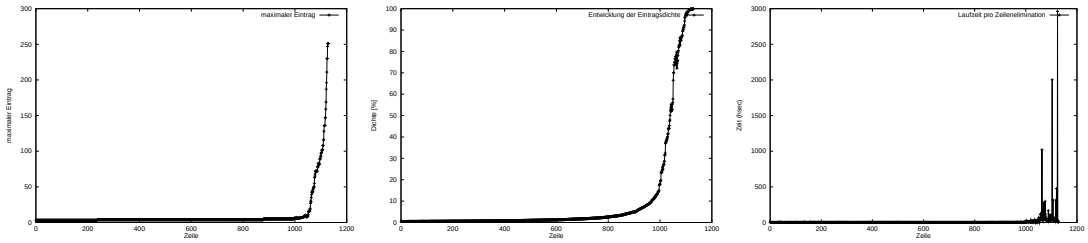
mgcd–Kennzahl: 14



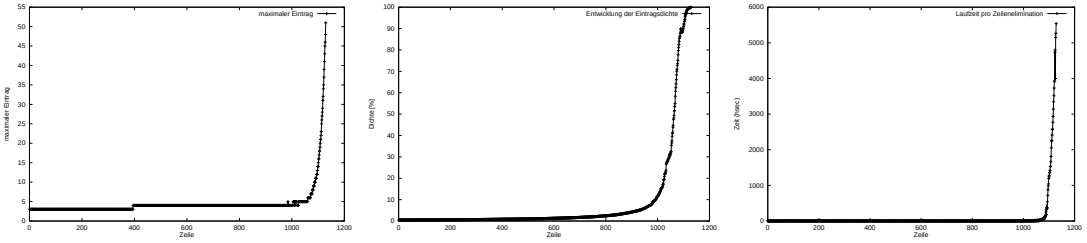
mgcd–Kennzahl: 15



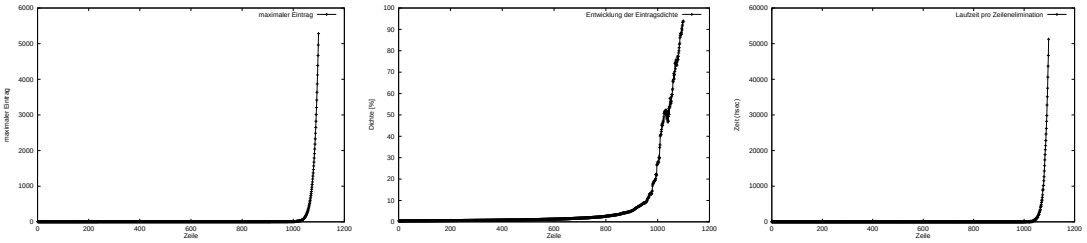
mgcd–Kennzahl: 16



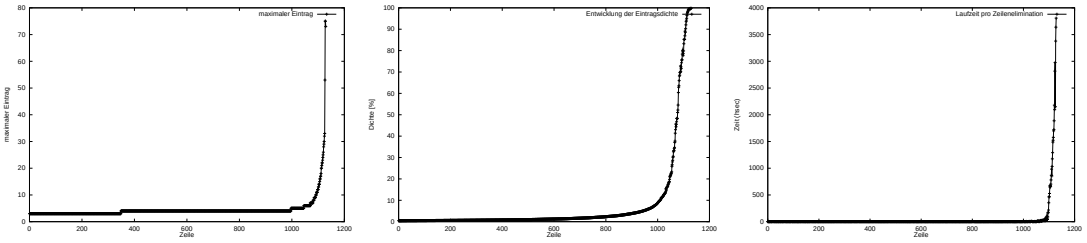
mgcd–Kennzahl: 17



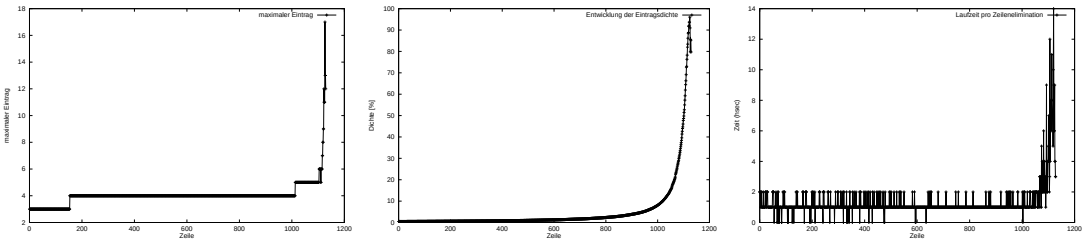
mgcd–Kennzahl: 18



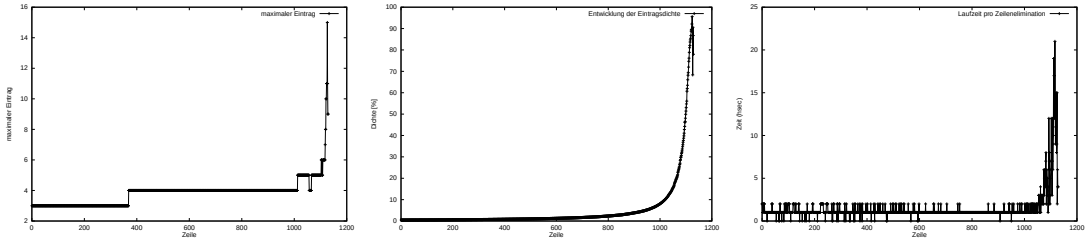
mgcd–Kennzahl: 19



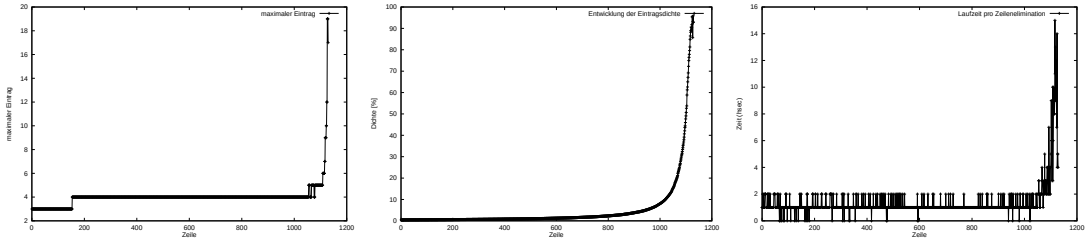
mgcd–Kennzahl: 20



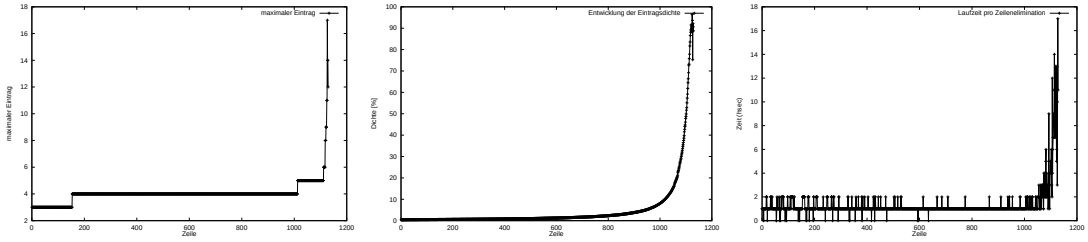
mgcd–Kennzahl: 21



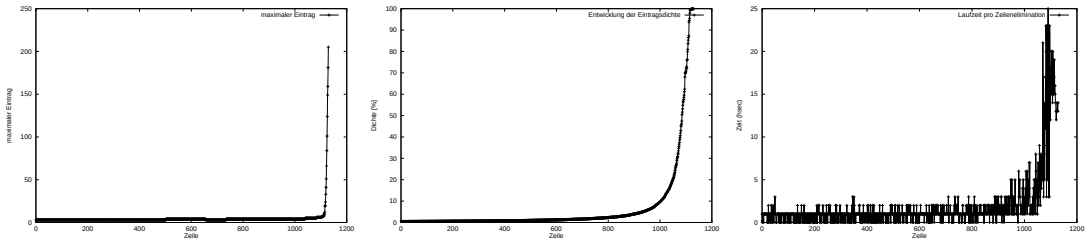
mgcd–Kennzahl: 22



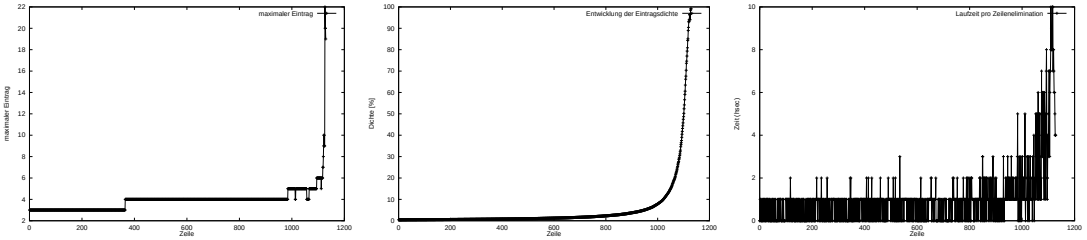
mgcd–Kennzahl: 23



mgcd–Kennzahl: 24



mgcd–Kennzahl: 25



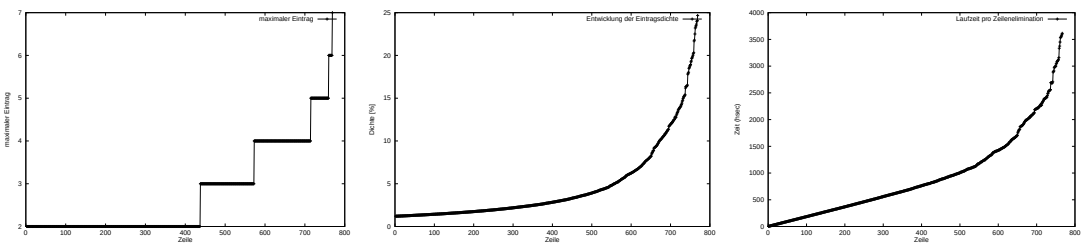
Anhang C

Ergebnisse: nichtmodulare, hauptminorenweise HNF–Berechnung

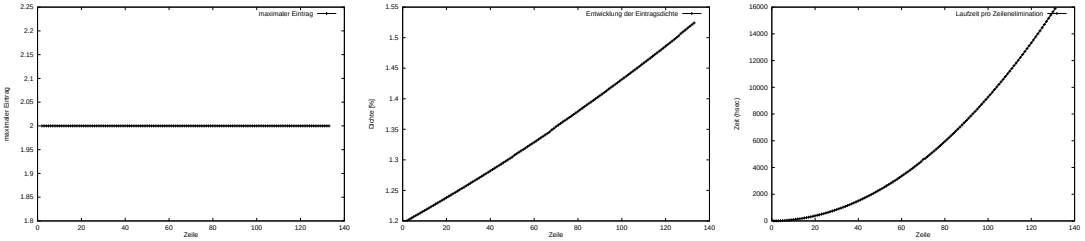
In diesem Kapitel haben wir die Laufzeitergebnisse der nichtmodularen HNF–Algorithmen, die sich einer hauptminorenweisen Vorgehensweise bedienen, für die Beispielmatrizen $BSP_{Jacobson}$ und BSP_{Neis} in Form von Grafiken zusammengestellt. Unser Ziel ist es, den Einfluß der *normalize*–Berechnung auf die Entwicklung des maximalen Eintrags, der Eintragsdichte und der Laufzeit pro Iteration zu ermitteln. Wir erhalten die folgenden Ergebnisse:

C.1 $BSP_{Jacobson}$

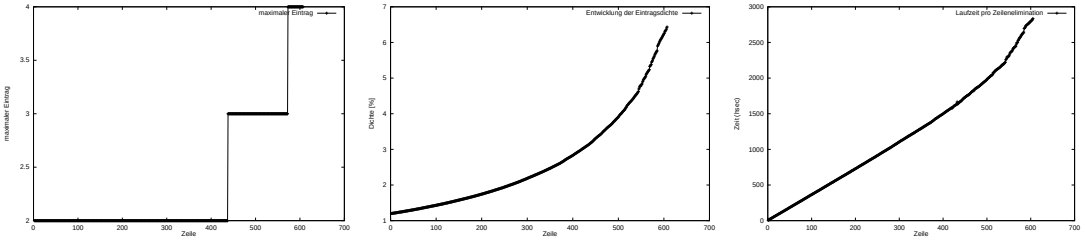
normalize–Kennzahl: 0



normalize–Kennzahl: 1

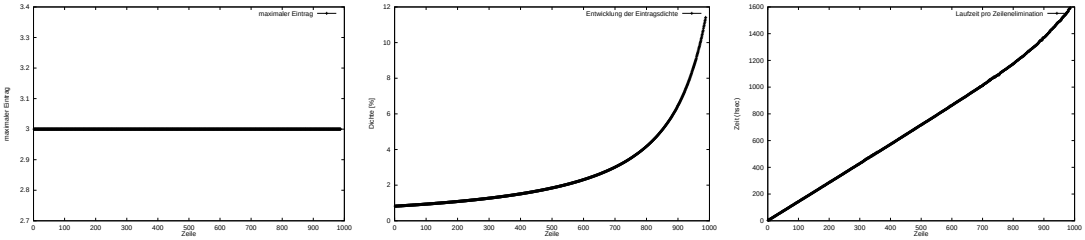


normalize–Kennzahl: 2

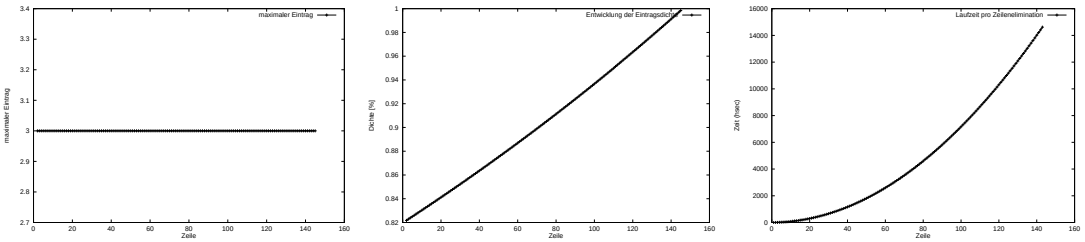


C.2 BSP_{Neis}

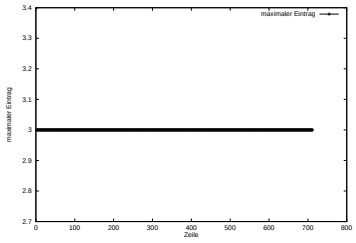
normalize–Kennzahl: 0

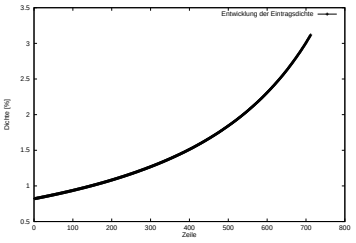


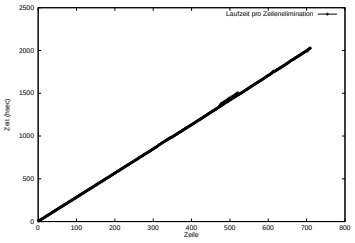
normalize–Kennzahl: 1



normalize–Kennzahl: 2







Literaturverzeichnis

[Bac98] W. Backes. Berechnung kürzester Gittervektoren. Diplomarbeit, Universität des Saarlandes, 1998.

[BBCO99] C. Batut, D. Bernardi, H. Cohen, and M. Olivier. *User's Guide to PARI-GP*. University of Bordeaux, 1999. available at ftp://megrez.math.u-bordeaux.fr.

[BBP95] I. Biehl, J. Buchmann, and T. Papanikolaou. LiDIA - A Library for Computational Number Theory. Technical report, Fachbereich Informatik, Universität des Saarlandes, 1995.

[BBT99] I. Biehl, J. Buchmann, and P. Theobald. Fast Probabilistic Computation of the Essential Part of the Hermite Normal Form of Sparse $(m \times m)$ Integer Matrices. 1999.

[Bla63] W. A. Blankinship. A new Version of the Euclidean Algorithm. *Amer. Math. Mon.*, (70):742–745, 1963.

[BM91] J. Buchmann and V. Müller. Algorithms for factoring integers. 1991.

[BM99] J. Buchmann and M. Maurer. Wie sicher ist die Public–Key–Kryptographie? Technical Report 99-02, Technische Universität Darmstadt, Darmstadt, 1999.

[Bra70] G. H. Bradley. Algorithm and Bound for the Greatest Common Divisor of n Integers. *Numerical Mathematics, ACM Transactions on Mathematical Software*, 13(7):433–436, July 1970.

[BS96] E. Bach and J. Shallit. *Algorithmic Number Theory, Efficient Algorithms*, volume 1. MIT Press, 1996.

[CC82] T.-W. J. Chou and G. E. Collins. Algorithms for the solution of systems of linear Diophantine equations. *SIAM Journal of Computation*, 11(4):687–708, November 1982.

[CLR89] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, McGraw–Hill Book Company, 1989.

[Coh93] H. Cohen. *A Course in Computational Algebraic Number Theory*. SV, New York, 1993.

[Den97] T. F. Denny. *Lösen großer dünnbesetzter Gleichungssysteme über endlichen Primkörpern*. Dissertation, Universität des Saarlandes, Saarbrücken, 1997.

[DKJ87] P. D. Domich, R. Kannan, and L. E. Trotter Jr. Hermite Normal Form Computation using Modulo Determinant Arithmetic. *Mathematics of Operations Research*, 12(1):50–59, February 1987.

[Dom89] P. D. Domich. Residual Hermite Normal Form Computations. *ACM Transactions on Mathematical Software*, 15(3):275–286, September 1989.

[Dü91] S. Düllmann. *Ein Algorithmus zur Bestimmung der Klassengruppe positiv definiter binärer quadratischer Formen.* Dissertation, Lehrstuhl Prof. Dr. Buchmann, Universität des Saarlandes, 1991.

[Fru77] M. A. Frumkin. Polynomial Time Algorithms in the Theory of Linear Diophantine Equations. In M. Karpinski, editor, *Lecture Notes in Computer Science 56*, pages 386–392, New York, 1977. Springer Verlag.

[Gro99] LiDIA Group. Lidia–Homepage der TU Darmstadt. http://www.informatik.tu-darmstadt.de/TI/LiDIA.html, 1999.

[Hav91] G. Havas. Coset enumeration strategies. In Stephen M. Watt, editor, *ISSAC'91*, pages 191–199, New York, 1991. ACM Press.

[Her51] C. Hermite. Sur l'indroduction des variables continues dans la theorie des nombres. *J. Reine Angew. Math.*, (41):191–216, 1851.

[HHR93] G. Havas, D. F. Holt, and S. Rees. Recognizing badly presented $\mathbb{Z}$-modules. In *Linear Algebra and its Applications*, number 192, pages 137–163, 1993.

[HM91] J. L. Hafner and K. S. McCurley. Asymptotically fast triangularization of matrices over rings. *SIAM J. COMPUT.*, 20(6):1068–1083, December 1991.

[HM94a] G. Havas and B. S. Majewski. Hermite normal form computation for integer matrices. *Congressus Numerantium*, (105):184–193, 1994.

[HM94b] G. Havas and B. S. Majewski. Integer matrix diagonalization. Technical Report TR0277, Key Centre for Software Technology, Department of Computer Science, The University of Queensland, 1994.

[HMM94] G. Havas, B. S. Majewski, and K. R. Matthews. Extended gcd algorithms. Technical Report TR0302, The University of Queensland, Brisbane, 1994.

[HS79] G. Havas and L. S. Sterling. *Integer Matrices and Abelian Groups.* 1979.

[Ili89a] C. S. Iliopoulos. Worst-Case Complexity Bounds on Algorithms for Computing the Canonical Structure of Finite Abelian Groups and the Hermite and Smith Normal Forms of an Integer Matrix. *SIAM J. COMPUT.*, 18(4):658–669, August 1989.

[Ili89b] C. S. Iliopoulos. Worst-Case Complexity Bounds on Algorithms for Computing the Canonical Structure of Infinite Abelian Groups and Solving Systems of Linear Diophantine Equations. *SIAM J. COMPUT.*, 18(4):670–678, August 1989.

[Jr.99] M. J. Jacobson Jr. *Subexponential Class Group Computation in Quadratic Orders.* Dissertation, Technische Universität Darmstadt, 1999.

[KB79] R. Kannan and A. Bachem. Polynomial Algorithms for Computing the Smith and Hermite Normal Forms of an Integer Matrix. *SIAM J. Computation*, 8(4):499–507, November 1979.

[Lam78] E. Lamprecht. *Einführung in die Algebra*. Birkhäuser Verlag, 1978.

[Lan52] C. Lanczos. Solution of systems of linear equations by minimized iterations. *J. Res. Nat. Bureau of Standards*, 49:33–53, 1952.

[LO91] B. A. LaMacchia and A. M. Odlyzko. Solving large sparse linear systems over finite fields. In *Advances in Cryptology – CRYPTO'90 (LNCS 537)*, pages 109–133, 1991.

[LSL99] L.-Q. Lee, J. G. Siek, and A. Lumsdaine. The Generic Graph Component Library. In *submitted OOPSLA*, 1999.

[Mag99] Magma–Homepage.
http://www.maths.usyd.edu.au:8000/comp/magma/Overview.html, 1999.

[Map99] Maple–Homepage.
http://www.maple.com, 1999.

[Mat99] Mathematica–Homepage.
http://www.mathematica.com, 1999.

[McC73] M. T. McClellan. The Exact Solution of Systems of Linear Equations with Polynomial Coefficients. *Journal Assoc. Comput. Machin.*, 20:563–588, 1973.

[MH94a] B. S. Majewski and G. Havas. The complexity of greatest common divisor computations. In *Algorithmic Number Theory, Lecture Notes in Computer Science 877*, pages 184–193, 1994.

[MH94b] B. S. Majewski and G. Havas. On the greatest common divisor of sets of integers. Technical Report TR0288, Key Centre for Software Technology, Department of Computer Science, The University of Queensland, 1994.

[MH95] B. S. Majewski and G. Havas. A solution to the extended gcd problem. In *Prodeedings of the 1995 International Symposium on Symbolic and Algebraic Computation*. ACM, ACM Press, New York, 1995.

[Mon95] P. L. Montgomery. A Block Lanczos Algorithm for Finding Dependencies over gf(2). In *Advances in Cryptology – Eurocrypt'95 (LNCS 921)*, pages 106–120, 1995.

[Mül94] A. Müller. Effiziente Algorithmen für Probleme der linearen Algebra über $\mathbb{Z}$. Diplomarbeit, Lehrstuhl Prof. Dr. Buchmann, Universität des Saarlandes, 1994.

[Nei94] S. Neis. Kurze Darstellung von Ordnungen. Diplomarbeit, Universität des Saarlandes, 1994.

[Nei01] S. Neis. *Zur Berechnung von Klassengruppen*. Dissertation, Technische Universität Darmstadt, 2001.

[New72] M. Newman. *Integral Matrices*. Pure and applied mathematics A series of monographs and textbooks. Academic Press, New York and London, 1972.

[Pap97] T. Papanikolaou. *Software-Entwicklung in der Computer-Algebra am Beispiel einer objektorientierten Bibliothek für algorithmische Zahlentheorie*. Dissertation, Universität des Saarlandes, 1997.

[PRL95] R. Pozo, K. A. Remington, and A. Lumsdaine. *SparseLib++ v. 1.5, Sparse Matrix Class Library, Reference Guide*, November 1995.

[SL98a] J. G. Siek and A. Lumsdaine. The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra. In *International Symposium on Computing in Object–Oriented Parallel Environments*, 1998.

[SL98b] J. G. Siek and A. Lumsdaine. The Matrix Template Library: A Unifying Framework for Numerical Linear Algebra. In *Parallel Object Oriented Scientific Computing*, 1998.

[SL98c] J. G. Siek and A. Lumsdaine. A Rational Approach to Portable High Performance: The Basic Linear Algebra Instruction Set (BLAIS) and the Fixed Algorithm Size Template (FAST) Library. In *Parallel Object Oriented Scientific Computing*, 1998.

[SLL98] J. G. Siek, A. Lumsdaine, and L.-Q. Lee. Generic Programming for High Performance Numerical Linear Algebra. In *Workshop on Interoperable Object–Oriented Scientific Computing*. SIAM, 1998.

[Sto96a] A. Storjohann. Computing Hermite and Smith Normal Forms of Triangular Integer Matrices. Technical Report 256, Department of Computer Science, ETH Zürich, Switzerland, 1996.

[Sto96b] A. Storjohann. A Fast+Practical+Deterministic Algorithm for Trinagularizing Integer Matrices. Technical Report 255, Department of Computer Science, ETH Zürich, Switzerland, 1996.

[Sto97] A. Storjohann. A Solution to the Extended GCD Problem with Applications. In *Proceeding of ISSAC'97*, 1997.

[Str92] B. Stroustrup. *Die C++ Programmiersprache*. Addison–Wesley, 2 edition, 1992.

[Str98] B. Stroustrup. *Die C++ Programmiersprache*. Addison–Wesley, 3 edition, 1998.

[The95] P. Theobald. Eine Implementierung der Linearen Algebra über $\mathbb{Z}$. Diplomarbeit, Lehrstuhl Prof. Dr. Buchmann, Universität des Saarlandes, 1995.

[Tis98] E. R. Tisdale. The C++ Scalar, Vector, Matrix and Tensor classes. August 1998.

[TWB97] P. Theobald, S. Wetzel, and W. Backes. Design Concepts for Matrices and Lattices in Lidia. In *Symposium on Applied Computing*, pages 532–536. ACM, 1997.

[vEB81] P. van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical Report MI/UVA 81-04, Mathematisch Instituut, Amsterdam, Amsterdam, 1981.

[Vel95a] T. L. Veldhuizen. Expression templates. *C++ Report*, 5(7):26–31, 1995.

[Vel95b] T. L. Veldhuizen. Using C++ Template Metaprograms. *C++ Report*, May 1995.

[Vel98] T. L. Veldhuizen. Techniques for Scientific C++. April 1998.

[VP96] T. L. Veldhuizen and K. Ponnambalam. Rapid Linear Algebra in C++. *Dr. Dobb's Journal*, August 1996.

[Wag97] C. Wagner. *Normalformberechnung von Matrizen über euklidischen Ringen.* Dissertation, Fachbereich Mathematik und Informatik der Universität/GHS Essen, November 1997.

[Web95] K. Weber. The Accelerated GCD Algorithm. *ACM Transactions on Mathematical Software*, 21(1):111–122, März 1995.

[Wet98] G. S. Wetzel. *Lattice Basis Reduction Algorithms and their Applications.* Dissertation, Universität des Saarlandes, 1998.

[Wie86] D. H. Wiedemann. Solving Sparse Linear Equations Over Finite Fields. In *IEEE Transactions on Information Theory*, volume IT-32, pages 54–62. IEEE, January 1986.

[WTB97] G. S. Wetzel, P. Theobald, and W. Backes. Lattice Reduction Algorithms and their Implementation. In Otto Spaniol, editor, *Promotion tut not: Innovationsmotor Graduiertenkolleg*, volume 21 of *Aachener Beiträge zur Informatik*, pages 65–85. Otto Spaniol, 1997.

Curriculum Vitae
(wissenschaftlicher Werdegang)

28.03.1970	geboren in Wadern (Saarland)
1976–1980	Grundschule Weiskirchen Konfeld
1980–1989	Staatliches Hochwald–Gymnasium Wadern *Abschluß: Allgemeine Hochschulreife*
10/1990–02/1996	Studium der Informatik Universität des Saarlandes *Abschluß: Diplom Informatiker*
04/1995–07/1996	Studentische Hilfskraft am Lehrstuhl Prof. J. Buchmann Universität des Saarlandes
08/1996–12/1998	Wissenschaftlicher Mitarbeiter am Lehrstuhl Prof. J. Buchmann Technische Universität Darmstadt